Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matej Žniderič

# Matrični algoritmi na podatkovno-pretokovnih računalnikih

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2016

*Zahvaljujem se doc. dr. Juriju Miheliču za strokovno pomoč in nasvete pri delu, Ivanu Milankoviću za pomoč pri izvajanju programske kode in Petri za pomoč pri lektoriranju.*

# Contents

# Seznam uporabljenih kratic

| kratica | angleško | slovensko |
|---|---|---|
| **APSP** | all-pairs shortest path | najkrajše poti med vsemi pari vozlišč |
| **CPE** | central processing unit | centralna procesna enota |
| **DFE** | dataflow engine | podatkovno-pretokovna enota |
| **DSP** | digital signal processor | digitalni signalni procesor |
| **FMem** | fast memory | hitri pomnilnik |
| **FPGA** | field programmable gate array | programabilno vezje |
| **LMem** | large memory | veliki pomnilnik |
| **SLiC** | Simple Live CPU interface | Maxelerjev vmesnik med CPE in podatokovno-pretokovno arhitekturo |

# Povzetek

**Naslov:** Matrični algoritmi na podatkovno-pretokovnih računalnikih

Medtem ko se frekvenca procesorjev že desetletje bistveno ne povečuje več, se potrebe znanosti po računski moči večajo. Podatkovno-pretokovna računalniška arhitektura predstavlja dobro alternativo klasičnemu ukazno-pretokovnemu računalniku. V okviru dela je bil razvit sklop algoritmov za množenje matrike z vektorjem in množenje dveh matrik na podatkovno-pretokovni arhitekturi. Primerjali so se časi izvajanja v primerjavi z ukazno-pretokovnimi rešitvami. Pohitritve pri množenju matrike in vektorja ni bilo. Pri množenju matrike z naborom vektorjem so bile dosežene skoraj 4-kratne pohitritve. Algoritem za množenje dveh matrik je dosegel več kot 100-kratno pohitritev. Dodatno sta bila implementirana algoritem za matrično potenciranje in algoritem iskanja najkrajših poti med vsemi pari vozlišč. Prvi je dosegel okoli 100-kratno pohitritev, drugi pa je bil približno enako hiter kot ukazno-pretokovni algoritem Floyd-Warshall.

**Ključne besede:** podatkovno-pretokovna arhitektura, matrično množenje, matrični algoritmi, grafni algoritmi, Maxeler Technologies

# Abstract

**Title:** Matrix algorithms on data-flow computers

While the processor frequency has failed to significantly increase in the last decade, the needs of science for computing power continue to increase. Data-flow computer architecture presents a good alternative to traditional control-flow computer. In this work a set of algorithms for matrix-vector multiplication and matrix-matrix multiplication was developed. It was compared with control-flow implementations. With matrix-vector multiplication there was no speedup. Algorithm for multiplying matrix with a set of vectors achieved almost 4-fold speedup. Matrix-matrix multiplication algorithm achieved more than 100-fold speedup. Further matrix exponentiation and all-pairs shortest path search algorithm were implemented. First algorithm achieved 100-times speedup, while the second one was about the same speed as control-flow Floyd-Warshall algorithm.

**Keywords:** dataflow architecture, matrix multiplication, matrix algorithms, graph algorithms, Maxeler Technologies

# Poglavje 1

# Uvod

Veliki napredki računalniške tehnologije so omogočili reševanje znanstvenih problemov, ki so bili prej izven dosega. Tako v raziskovalnem delu kot v gospodarstvu, so se začele uporabljati metode znanstvenega računanja, ki probleme rešujejo z numeričnimi metodami, izdelavo napovednih modelov in s simulacijami. Pri problemih tega področja računska moč definira, katere probleme je mogoče rešiti in kako natančni so lahko rezultati. Zaradi želj po reševanju novih in bolj zahtevnih problemov se potrebe po zmogljivosti računalnikov iz dneva v dan večajo. Po drugi strani se je frekvenca procesorjev v večji meri nehala povečevati in Moorov zakon že več kot desetletje ne deluje več. Na trg so prišli več jedrni procesorji, ki pa imajo še vedno veliko omejitev. V osnovi so paralelni, v resnici pa si delijo veliko sredstev in v splošnem še vedno niso dovolj zmogljivi [24, 27].

Problem lahko deloma rešimo tako, da se s splošnonamenske arhitekture premaknemo na problemsko specifično arhitekturo. Dobra alternativa običajni ukazno-pretokovni arhitekturi je podatkovno-pretokovna arhitektura. Za vrsto problemov je lahko hitrejša, energetsko bolj učinkovita in fizično manjša od ukazno-pretokovne arhitekture. Primerna je predvsem na področju znanstvenega računanja in pri delu z masovnimi podatki [25].

Eden izmed računsko zahtevnih problemov, ki bi ga bilo dobro pohitriti, je matrično množenje. V magistrskem delu ga bomo poskušali pohitriti s pre-

nosom na podatkovno-pretokovni računalnik podjetja Maxeler Technologies. Problem se pojavlja na najrazličnejših področjih, zaradi česar bo zanimiv za široko množico ljudi. Veliko dela je bilo že vloženega v manjšanje asimptotične časovne zahtevnosti (npr. v [30, 10]). Ta je v teoriji sicer dobra, v praksi pa šteje predvsem realni čas izvajanja [20]. Zato bomo raje vzeli običajni algoritem množenja in poskušali prednosti doseči pri bolj učinkoviti arhitekturi. Pri tem bomo uporabljali tehnike eksperimentalne algoritmike, ki kombinira teorijo algoritmov z eksperimenti [18, 23]. Tako bomo za vsak problem, namesto enega razvili sklop algoritmov in jih eksperimentalno ovrednotili. Najboljše bomo primerjali še z ukazno-pretokovno različico in s tem raziskali, ali je prenos naših algoritmov na podatkovno-arhitekturo pravzaprav sploh smiseln.

Nekaj dela na temo matričnega množenja na računalniku Maxeler je bilo že narejenga. V delu [21] so implementirali dve različici in dosegali do 18-kratne pohitritve v primerjavi z ukazno-pretokovnim algoritmom. Med pisanjim magistrske naloge je bila v [29] predstavljena še ena rešitev. Računali so na redkih matrikah in dosegli okoli 10-kratne pohitritve. Po našem mnenju je problem kljub temu še vedno premalo raziskan. V našem delu bomo predstavili nekaj novih algoritmov matričnega množenja in odgovorili na vprašanja, kot so: kako število cevi in uporaba velikega pomnilnika (LMema) vplivata na hitrosti izvajanja. Izvedli bomo obsežno eksperimentalno ovrednotenje. Idealno bi bilo doseči tudi dodatne pohitritve. Dodatno si bomo pogledali problem množenja matrike in vektorja ter problem množenja z naborom vektorjev. Tokrat bo tudi prvič raziskana ideja uporabe matričnega množenja in računalnika Maxeler za reševanje grafnih algoritmov.

Delo bomo razdelili po poglavjih na naslednji način. V poglavju 2 bomo začeli z opisom ukazno-pretokovne in podatkovno-pretokovne računalniške arhitekture. Videli bomo, da se med seboj zelo razlikujeta, ter predstavili prednosti in slabosti posamezne arhitekture. Pogledali si bomo podatkovno-pretokovni računalnik podjetja Maxeler Technologies. Na koncu poglavja bomo naredili kratek pregled pogojev za doseganje pohitritev algoritmov in

načine, kako jih lahko dosežemo. V poglavju 3 bomo implementirali sklop algoritmov za reševanje problema množenja matrike in vektorja. Začeli bomo z osnovnim algoritmom in ga postopoma pohitrili. Predstavili bomo dva načina povečanja paralelizacije: z boljšim izkoristkom cevovoda in z uporabo cevi. Implementiran bo tudi algoritem za množenje z naborom vektorjev. Vse algoritme bomo eksperimentalno ovrednotili. V poglavju 4 se bomo lotili matričnega množenja. Za osnovo bomo vzeli algoritme množenja matrike in vektorja ter jih nadgradili v matrično množenje. Nadaljnje pohitritve bomo dosegali z uporabo LMema in dodatnim večanjem števila cevi. Posebno zanimiv je zadnji algoritem, ki matriko razdeli na ploščice in z njimi učinkovito računa. Tudi te algoritme bomo eksperimentalno ovrednotili. V poglavju 5 bo predstavljen sklop sorodnih matričnih algoritmov, ki jih je mogoče implementirati z matričnim množenjem in s tem prenesti na podatkovno-pretokovno arhitekturo. Nato bomo izbrali najhitrejši algoritem matričnega množenja ter ga nadgradili v algoritem za potenciranje matrik in algoritem iskanja najkrajše poti med vsemi pari vozlišč. Prvega bomo primerjali z ukazno-pretokovno različico istega algoritma, slednjega pa z algoritmom Floyd-Warshall. Sledil bo še zaključek.

# Poglavje 2

# Podatkovno-pretokovna arhitektura

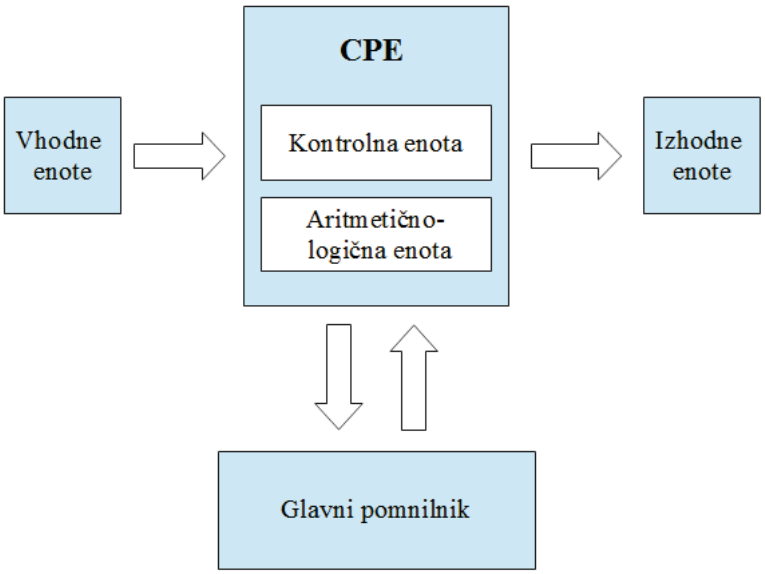
## 2.1 Ukazno-pretokovna arhitektura

Najpogostejša arhitektura računalnikov v današnjem času je ukazno-pretokovna arhitektura. V začetku razvoja računalništva je bilo modelov z ukazno-pretokovno arhitekturo več (npr. Harvardska arhitektura), v veliki večini pa se je obdržala von Neumannova arhitektura oziroma von Neumannov računalniški model. Z njim se je pojavil koncept shranjenega programa, ki je ločen od strojne opreme. Bistveni so trije osnovni deli, ki model sestavljajo: centralna procesna enota (CPE), glavni pomnilnik in vhodno-izhodne enote. Model se je izkazal za tako uporabnega, da je bil naslednjih nekaj desetletij praktično edini računalniški model [7].

Zgradbo von Neummanovega računalnika vidimo na sliki 2.1. Delovanje lahko povzamemo na naslednji način:

1. Izvajanje določa zaporedje ukazov, ki mu pravimo program. Ukazi se izvajajo zaporedno eden za drugim.

2. Računalnik ukaz izvede tako, da ga najprej prebere iz pomnilnika. Zatem naloži ustrezne podatke in v CPE izvrši ustrezno operacijo nad




Slika 2.1: Von Neumannov računalniški model

njimi. Na koncu rezultat operacije zapiše nazaj v pomnilnik.

3. Programski števec, ki kaže na naslednji ukaz za izvedbo, se po izvedbi ukaza poveča za ena. V primerih, ko gre za skočne ukaze, se programski števec nastavi na drugo ustrezno vrednost.

4. Vhodne in izhodne enote računalnik uporablja za komunikacijo z zunanjim svetom [7].

V začetku razvoja računalništva so bili računalniki večji kot danes. Eden izmed glavnih ciljev von Neumannovega modela je bil minimizacija strojne opreme. Sčasoma so se potrebe spreminjale in je postajala vedno bolj pomembna čim večja hitrost. Za čim hitrejši računalnik zaporedna arhitektura ni nujno optimalna. Operacije se izvajajo zaporedno ena za drugo, kar pomeni, da je hitrost omejena s časom izvedbe operacije. Na koncu vsake operacije se mora rezultat zapisati in nato brati iz pomnilnika tudi v primerih, ko je rezultat potreben za naslednjo operacijo. Moderni procesorji te probleme poznajo in jih rešujejo na različne načine; višja frekvenca, cevovod,

večnivojska predpomnilniška hierarhija in uporaba predikcije skočnih ukazov jih naredi veliko bolj učinkovite. Kljub temu je model po naravi zaporeden in ima svoje omejitve [26].

Zakaj se torej ta model danes še vedno uporablja? Odgovorov je več. Eden izmed razlogov je, da je za probleme, kjer ni veliko možnosti za izkoriščanje paralelizma, von Neumannova arhitektura lahko še vedno najprimernejša. Ločenost CPE in programa tudi danes zmanjša velikost strojne opreme. Višjenivojski programski jeziki, kot so C, java in python, omogočajo hitro in učinkovito programiranje. V času masovnih podatkov in računsko zahtevnih problemov te prednosti niso več dovolj. Za take probleme je namreč vsakokratno branje in pisanje v pomnilnik ter ena enota (ali več v primeru več jeder) za izvedbo računskih operacij premalo in je arhitektura neučinkovita [26, 33].

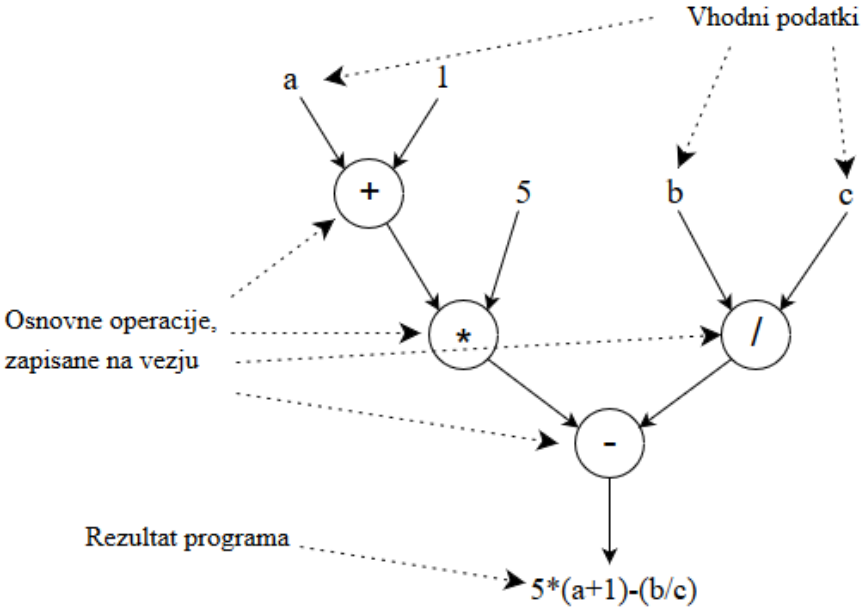
## 2.2 Podatkovno-pretokovna arhitektura

Obstaja več alternativ von Neummanovi arhitekturi računalnika. Najbolj različna od von Nemmanove je podatkovno-pretokovna (angl. dataflow) arhitektura. Za vrsto problemov, predvsem na področju znanstvenega računanja in pri delu z masovnimi podatki, je lahko veliko hitrejša, energetsko bolj učinkovita in fizično manjša od von Neumannove arhitekture [25].

Podatkovno-pretokovna arhitektura ne pozna CPE, programskega števca in ukazov. Program je (po prevajanju) zapisan direktno na vezju v obliki podatkovno-pretokovnega grafa, kjer so vozlišča grafa osnovne računske operacije (npr. seštevanje ali množenje), povezave pa predstavljajo odvisnost operacij med seboj [26]. Zaradi preprostosti osnovnih operacij jih je lahko na čipu veliko. Potek programa je določen z razpoložljivostjo operandov; ko so na voljo vsi operandi, potrebni za izvedbo operacije, se ta izvede in pošlje naprej rezultat. Podatki se pretakajo iz pomnilnika na čip, kjer se posredujejo od enega vozlišča do drugega brez vmesnega dostopanja do zunanjega pomnilnika. Tudi kompleksnejši izračuni se lahko v celoti izvedejo na

čipu. Do pomnilnika se ponovno dostopa šele ob pisanju končnega rezultata [7]. Za razliko od von Neummanovega računalnika, kjer se operacije izvajajo v različnih časih na isti funkcionalni enoti (na CPE), je pri podatkovno-pretokovnih računalnikih celoten izračun razporejen na čipu. Celoten izračun je cevovod (podatkovni konflikti se rešijo že med prevajanjem), ki lahko vsako urino periodo vrne rezultat [25].

Skupku operacij, povezanih v graf, pravimo ščepec. Primer preprostega izračuna vidimo na sliki 2.2. Spremenljivke $a$, $b$ in $c$ predstavljajo vhodne podatke. Osnovne računske operacije, konstanti 1 in 5 ter usmerjene povezave sestavljajo program, ki je zapisan na vezju. Rezultat programa je odvisen od vhodnih podatkov, na podlagi katerih program vrne rezultat aritmetičnega izraza $5 * (a + 1) - (b/c)$.




Slika 2.2: Graf ščepca, ki izračuna vrednost $5 * (a + 1) - (b/c)$

Podatkovno-pretokovni računalniki so že po definiciji paralelni. Istočasno se lahko izračunava na tisoče osnovnih računskih operacij. Prispodoba za prehod z ukazno-pretokovne na podatkovno-pretokovno arhitekturo je uvedba

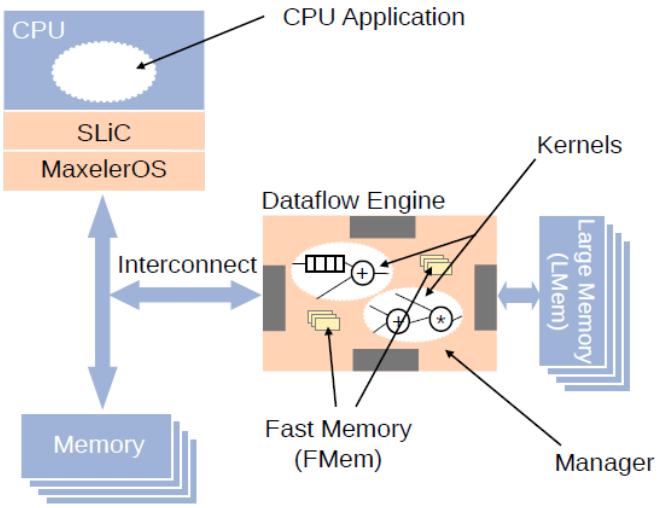
Fordovega tekočega traka. Namesto majhnega števila dragih in počasnih ekspertov (v našem primeru ena ali več CPE), ki znajo vse, imamo veliko število specializiranih delavcev (v našem primeru osnovne računske operacije), ki zelo dobro opravljajo svojo nalogo. Proces postane hitrejši in cenejši. Uvedba tekočega traka zahteva nekaj načrtovanja in je seveda smiselna pri večjem številu enakih izdelkov (v našem primeru enakih izračunov na različnih podatkih).

## 2.3 Strojna oprema računalnika Maxeler

V magistrski nalogi bomo uporabljali opremo podjetja Maxeler Technologies. Podjetje ponuja kompletno programsko in strojno opremo za podatkovno-pretokovno računanje. Maxeler Technologies sodeluje z velikimi podjetji s področja nafte in zemeljskega plina, finančne analize, visokofrekvenčnega trgovanja in znanstvenega računanja [17]. Za lažje razumevanje naših rešitev je treba razumeti osnovno arhitekturo našega sistema.

Pri strojni opremi Maxeler gre za kombinacijo običajne CPE in programabilnega vezja FPGA (Field Programmable Gate Array), na katerega prenesemo del aplikacije, ki ga želimo pospešiti. Zgradbo prikazuje slika 2.3. Poleg CPE in vezja FPGA prikazuje tudi tri vrste pomnilnika. Imamo pomnilnik običajnega računalnika (na sliki Memory), ki ga uporablja CPE. Pomnilnika FMem (hitri pomnilnik) in LMem (veliki pomnilnik) komunicirata direktno s podatkovno-pretokovno enoto. Pri razvoju algoritmov je ključnega pomena dobro načrtovanje uporabe pomnilnika. FMem nam omogoča hranjenje manjše količine (nekaj megabajtov) direktno na čipu in hiter, naključni dostop, LMem pa nam omogoča uporabo velikih količin (nekaj gigabajtov) podatkov izven čipa. Bolj podrobno uporabo pomnilnika bomo videli v naslednjih poglavjih.

Oprema Maxeler je na voljo v različnih oblikah. Za namene raziskovanja in razvoja aplikacij je za podatkovno-pretokovni del na voljo razširitvena kartica, ki jo je mogoče vstaviti v osebni računalnik. Za namene uporabe v




Slika 2.3: Arhitektura računalnika Maxeler [16]

industriji je na voljo več različic celotne opreme, ki so kombinacija ukazno-pretokovnega računalnika in večjega števila kartic. Razlike med različicami so v optimizaciji za določeno vrsto problemov. Podjetje ponuja opremo tudi v oblaku MaxCloud.

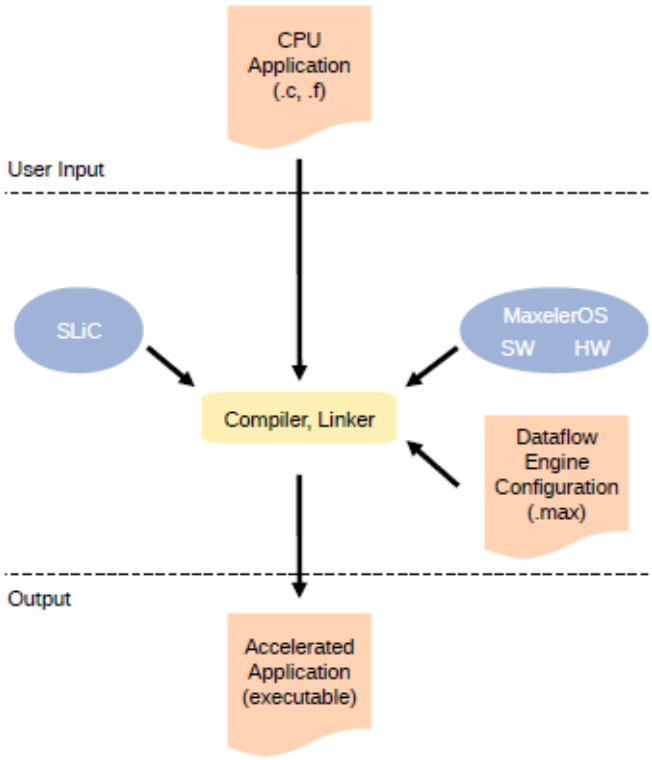
## 2.4 Programski paket MaxCompiler

Programski paket MaxCompiler je skupek orodij za razvoj in prevajanje programske opreme za računalnik Maxeler. Aplikacijo za računalnik Maxeler sestavljajo trije ločeni programi [13]:

- ščepec
- nadzornik
- ukazno-pretokovni program

Zgradbo MaxCompilerja prikazuje slika 2.4. Za prenos podatkov med CPE in podatkovno-pretokovno enoto se uporablja vmesnik imenovan SLiC.

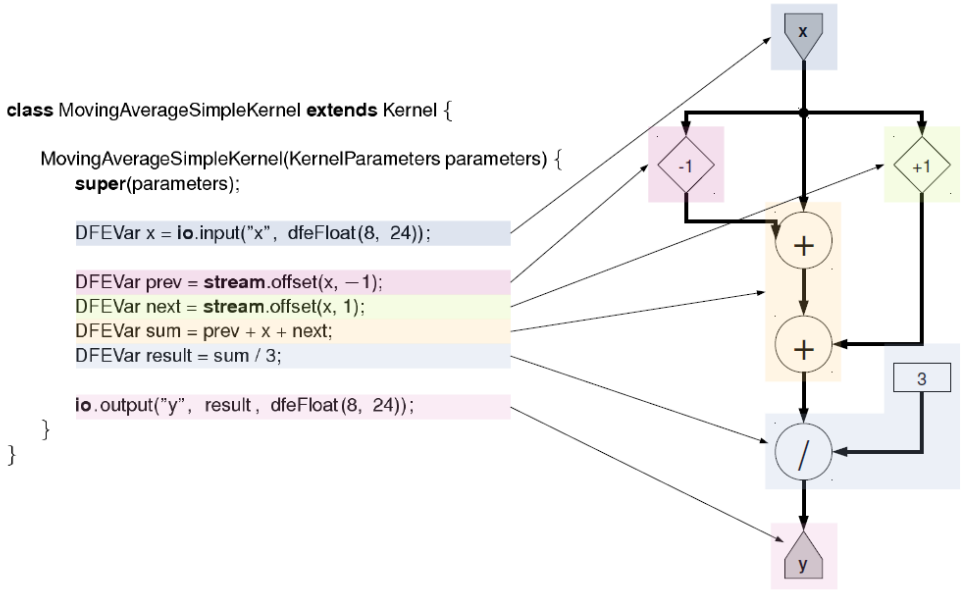
Paket vključuje razvojno okolje MaxIDE, ki temelji na Eclipsu, simulator in razhroščevalnik. S tem nam ponuja podporo pri celotnem razvoju programske opreme. Prevajalnik v celoti poskrbi za prenos programa na čip [13].




Slika 2.4: Zgradba MaxCompilerja [16]

### 2.4.1 Ščepec

Ščepci so deli programa, ki jih iz ukazno-pretokovnega dela prenesemo na podatkovno-pretokovni del in jih na ta način pospešimo. Programiranje ščepcev poteka v programskem jeziku MaxJ, ki je razširjena verzija jave. Pri razvoju ščepcev gre pravzaprav za programiranje predstavitev osnovnih računskih operacij in povezav med njimi v podatkovno-pretokovnem delu računalnika [14].




Slika 2.5: Ščepec za izračun drseče aritmetične sredine skupaj s pripadajočim grafom [16]

Slika 2.5 prikazuje kodo in graf preprostega ščepca za izračun drseče aritmetične sredine. Koda opisuje podatkovno-pretokovni graf, ki ga vidimo na desni strani slike. Ob zagonu v ščepec pošljemo vrsto podatkov, ki tečejo skozi graf eden za drugim in nad katerimi se izvajajo izračuni. Ob programiranju ščepca tako ne pišemo same zanke, ampak le njeno telo (izračun). Pomen samih ukazov bomo videli v naslednjem poglavju.

### 2.4.2 Nadzornik

Nadzornik (angl. manager) prav tako uporablja jezik MaxJ in se na koncu prevede v strojno kodo. Definira, kako so ščepci povezani med seboj in skrbi za prenos podatkov med ukazno-pretokovnim in podatkovno-pretokovnim delom. Kontrolira tudi proces prevajanja kode. Možni sta osnovna in napredna konfiguracija. Slednja nam omogoča uporabo več ščepcev ter zahtevnejše interakcije med ščepci ter vhodno-izhodnimi enotami [14, 12].

### 2.4.3 Ukazno-pretokovni program

Oprema Maxeler oprema kombinira običajno ukazno-pretokovno enoto in podatkovno-pretokovno enoto. Ukazno-pretokovni program je ponavadi napisan v programskem jeziku C/C++ ali fortran. Tu se inicializirajo podatki in naredi razpored zaganjanja ščepcev. Do njih dostopamo preko vmesnika SLiC, ki nam omogoča sinhrono in asinhrono zaganjanje [14]. V ukazno-pretokovnem delu se izvaja tudi koda, ki je ni smisleno prenesti na podatkovno-pretokovno arhitekturo.

## 2.5 Pospeševanje algoritmov

Pohitritve algoritmov, prenesenih na arhitekturo Maxeler, so zelo različne. Algoritmi lahko dosežejo več kot 200-kratno pohitritev ali pa so počasnejši od običajnega ukazno-pretokovne arhitekture. V [27] so naredili študijo in ugotovili, da so za pohitritve bistveni:

**Masovni podatki.** Glavna lastnost podatkovno-pretokovnega računalnika Maxeler je pospeševanje premikov podatkov. Da se bodo pohitritve dejansko občutile, so najbolj primerne aplikacije, ki delajo z masovnimi podatki. Glavne pohitritve se skrivajo znotraj čipa, ko podatki potujejo skozi podatkovno-pretokovni graf. Čim daljši kot je graf, večje pohitritve lahko pričakujemo. Poleg tega ima računalnik Maxeler nekaj začetne zakasnitve, zaradi česar je treba imeti veliko podatkov, da začetna zakasnitev bistveno ne upočasni izvajanja in se lahko dosegajo velike pohitritve.

**Ponovna uporaba podatkov.** Aplikacija mora uporabiti podatke več kot enkrat. Prenos podatkov med ukazno-pretokovnim in podatkovno-pretokovnim delom zahteva čas, zato se večje pohitritve dosegajo pri večkratni uporabi podatkov.

**Struktura zank.** Dobro je, da programi vsebujejo zanke, ki porabijo večino izvajalnega časa. Zanke (ki upravljajo z masovnimi podatki ali de-

lajo kompleksne izračune) so tisti deli aplikacije, ki jih prenesemo na podatovno-pretokovni del računalnika in jih poskušamo pohitriti. Če se ravnamo po Amdahlovem zakonu, to pomeni, da če zanke vzamejo 90 % časa, so lahko pohitritve največ desetkratne. Če zanke zavzamejo 99 % časa, je teoretično možno doseči stokratno pohitritev.

**Začetna zakasnitev.** Algoritem mora tolerirati začetno zakasnitev. Podatki morajo za prihod prvega rezultata potovati skozi celoten čip s frekvenco, ki je nižja od običajne CPE (trenutno okoli 200 MHz), kar vzame čas

$$t = NT,$$

kjer je $N$ število vzporednih premikov znotraj zanke in $T$ čas cikla. Vsi naslednji rezultati pridejo s hitrostjo, ki je lahko veliko hitrejša od CPE [27].

Za hitrost je ključnega pomena način implementacije, kar bo se pokazalo tudi v naših algoritmih. Za doseg maksimalne pohitritve se je treba lotiti naslednjih področij:

**Koreografija podatkov.** Načrtovanje koreografije podatkov je ključnega pomena. Podatkovno-pretokovni računalnik ne pozna vnaprejšnjega branja (angl. prefetching), ki poskuša napovedati naslednji podatek ali ukaz in s tem pohitriti program. Naslednji ukaz je tisti, ki je na vezju za trenutnim ukazom, in naslednji podatek tisti, ki ga je programer smiselno postavil za trenutnim podatkom. V analizo podatkov in njihovo koreografijo je treba vložiti veliko truda in jo s tem čim bolj prilagoditi podatkovno-pretokovni arhitekturi.

**Izkoriščanje cevovoda.** Za doseg prave paralelizacije je treba razumeti koncept cevovoda. Program in tok podatkov je treba zapisati tako, da je čim manj čakalnih period, da je cevovod čimbolj zapolnjen in da v idealnem primeru program vsako urino periodo vrne en rezultat. Za doseg tega je pogosto treba spremeniti vrstni red ukazov in vhodnih podatkov.

**Analiza podatkov in predstavitev s plavajočo vejico.** Za največje pohitritve je treba optimizirati aritmetične operacije in velikost podatkov pri izračunih. Arhitektura je fleksibilna in omogoča določanje natančnosti v vsakem koraku izračuna. Velikost podatkov bo vplivala tako na hitrost izračuna kot na velikost programa na vezju. Večja kot je natančnost, več bo uporabljenih logičnih elementov in manj bo prostora za drugo kodo. Treba je razmisliti, kakšno natančnost dejansko potrebujemo, in nato optimizirati program.

**Različni algoritmi.** Kakšno stopnjo pohitritve bomo lahko dosegli, je pogosto težko napovedati. Če z zgornjimi izboljšavami ne pridemo do želene pohitritve, poskušamo na podatkovno-pretokovno arhitekturo prenesti drug algoritem, ki rešuje isti problem. Z metodo poskusov in napak na koncu izberemo najhitrejši algoritem [27, 22].

# Poglavje 3

# Množenje matrike in vektorja

Množenje matrike in vektorja je ena izmed najosnovnejših in ključnih operacij v računalništvu. Pri znanstvenem računanju so matrike pogosto velike, zato so učinkoviti algoritmi za množenje matrike in vektorja bistvenega pomena. Za matriko velikosti $n \times n$ je že zaradi same velikosti potrebnega $\Omega(n^2)$ časa. V primeru velikih matrik lahko čas izračuna hitro postane ovira. Zato bomo algoritem za množenje matrike in vektorja prenesli z običajnega ukazno-pretokovnega računalnika na podatkovno-pretokovni računalnik in ga poskušali pohitriti.

Poleg tega bo prenos algoritma služil tudi kot prikaz razvoja algoritmov na podatkovno-pretokovni arhitekturi. Pri razvoju za računalnik Maxeler gre za programiranje na veliko nižjem nivoju, kot je to pri običajnem razvoju za ukazno-pretokovni računalnik. Programi se izvajajo vzporedno na najnižjem nivoju in zahtevajo popolnoma nov način razmišljanja. Pri takih problemih je smiseln postopen pristop k programiranju. Zgledovali se bomo po [11], kjer so na postopen način pospešili algoritem, ki sešteje vrstice v matriki. V poglavju, ki sledi, se bomo lotili množenja dveh matrik; program za množenje matrike in vektorja nam bo služil kot dobra osnova za nadaljnje delo.

## 3.1 Definicija problema

Vzemimo matriko $A = [a_{i,j}]$ z dimenzijo $m \times n$, kjer je $m$ število vrstic in $n$ število stolpcev, ter vektor $B = [b_i]$ z dimenzijo $m$. Rezultat množenja $A \times B$ je vektor $C = [c_i]$ velikosti $m$ in elementi

$$c_i = \sum_{k=1}^{n} a_{i,k} \cdot b_k, 1 \leq i \leq m.$$

Množenje lahko predstavimo tudi v naslednji obliki:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_{11} \cdot b_1 + a_{12} \cdot b_2 + \dots + a_{1n} \cdot b_n \\ a_{21} \cdot b_1 + a_{22} \cdot b_2 + \dots + a_{2n} \cdot b_n \\ \vdots \\ a_{m1} \cdot b_1 + a_{m2} \cdot b_2 + \dots + a_{mn} \cdot b_n \end{bmatrix}$$

Razvidno je, da se bodo elementi matrike uporabili samo enkrat, elementi vektorja pa $m$-krat in bi jih bilo lahko smiselno hraniti na vezju. Rezultati končnih elementov med seboj niso povezani, kar pomeni, da bo izračune možno paralelizirati. Izračunajmo še število operacij. Za izračun enega elementa vektorja je treba zmnožiti vse elemente vrstice z vsemi elementi vektorja ($n$ množenj). Za celoten vektor je to treba ponoviti za $m$ vrstic. Število operacij množenja je

$$T^*_{vec}(m, n) = mn = \Theta(mn).$$

V vsaki vrstici je treba zmnožke sešteti ($n - 1$ seštevanj) in ponoviti za $m$ vrstic. Število operacij seštevanja je

$$T^+_{vec}(m, n) = mn - m \sim mn = \Theta(mn).$$

Skupno število operacij za izračun množenja matrike in vektorja je

$$\begin{aligned} T_{vec}(m, n) &= T^*_{vec}(m, n) + T^+_{vec}(m, n) \\ &= mn + mn - m \sim 2mn \\ &= \Theta(mn). \end{aligned}$$

## 3.2 Ukazno-pretokovna implementacija

Najprej smo razvili program za običajen ukazno-pretokovni računalnik. Služil nam bo za primerjavo časov s kasnejšimi pospešenimi rešitvami. Hkrati se bo uporabljal tudi za testiranje pravilnosti rešitev za računalnik Maxeler. Pri praktičnih implementacijah smo se zaradi enostavnejšega eksperimentalnega ovrednotenja omejili na kvadratne matrike. Program je naslednji:

```
1 void mulMatVec(int n, float *mat, float *vec, float *res){
2   for (int i = 0; i < n; i++) {
3     float sum = 0;
4     for (int j = 0; j < n; j++)
5       sum += mat[i * n + j] * vec[j];
6     res[i] = sum;
7   }
8 }
```

Program sprejme vhodne podatke $n$ (velikost matrike), *mat* (matrika velikosti $n \times n$), *vec* (vektor velikosti n), in *res* (spremenljivka, kamor se shrani rezultat). Vsi naši programi podpirajo štiri stikala:

- **-h, --help** izpiše navodila za uporabo
- **-n, --size** nastavimo velikost vhodne matrike
- **-r, --range** nastavimo razpon velikosti elementov
- **-t, --trace** ponuja opcijo 0 za izpis časa izvajanja, 1 za izpis celotnih vhodnih in izhodnih matrik ter 2 za preverjanje pravilnosti programa

Oglejmo si primer izvedbe množenja matrike velikosti $8092 \times 8092$ z vektorjem velikosti 8092, kjer so elemeti v razponu od $-200$ do 200 in je opcija za sledenje 0:

```
$ ./matvec --size 8092 -r 200 --trace 0
8092 25046424.00 89 80
```

Progam vrne štiri vrednosti: velikost matrike in vektorja $n$, vsoto izhodnega vektorja (za lažje delno preverjanje pravilnosti), realni čas izvajanja (ta nas

bo najbolj zanimal) in čas izvajanja na CPE. Prikazan primer zagona programa bomo (pri različno velikih vhodih) uporabljali med eksperimentalnim ovrednotenjem na koncu poglavja.

## 3.3 Programiranje nadzornika

Za povezavo med podatkovno-pretokovnim delom in ukazno-pretkovnim delom bomo v naslednjih poglavjih uporabljali nadzornika, ki bo skrbel za komunikacijo z vhodno-izhodnimi enotami in prenašanje podatkov med ukazno-pretokovnim in podatkovno-pretokovnim delom računalnika. Med različnimi implementacijami bodo samo manjše razlike, ki bodo navedene sproti. Primer nadzornika iz razdelka 3.4 za algoritem osnovnega množenja matrike in vektorja je naslednji:

```
1  class MultiplyManager{
2     static final int nMax = 20*1024;
3
4     public static void main(String[] args) {
5        Manager manager = new Manager(new EngineParameters(args));
6        Kernel kernel = new MultiplyKernel(manager.makeKernelParameters(),
             nMax);
7        manager.setKernel(kernel);
8        manager.setIO(IOType.ALL_CPU);
9        manager.createSLiCinterface(interfaceDefault());
10       manager.build();
11    }
12    ...
13 }
```

Koda prikazuje metodo *main*, ki sproži proces prevajanja. Metoda začne z inicializacijo objekta *manager* tipa *Manager*, ki predstavlja nadzornika. Ali se bo prevajalo za simulacijo ali podatkovno-pretokovni računalnik, je določeno z objektom *EngineParameters*. Zatem sledi inicializacija ščepca *kernel*, ki bo predstavljena v nadaljevanju. V 8. vrstici naštejemo tipe vhodno-izhodnih enot. V tem primeru je to samo CPE (kasneje bomo upo-

rabljali tudi direktno komunikacijo z LMemom). 9. vrstica izdela vmesnik, preko katerega lahko ukazno-pretokovni del dostopa do podatkovno-pretokovnega dela. Na koncu pokličemo metodo *build*(), ki začne prevajanje za simulator ali podatkovno-pretokovno enoto.

Poglejmo si še kodo za izdelavo vmesnika za CPE:

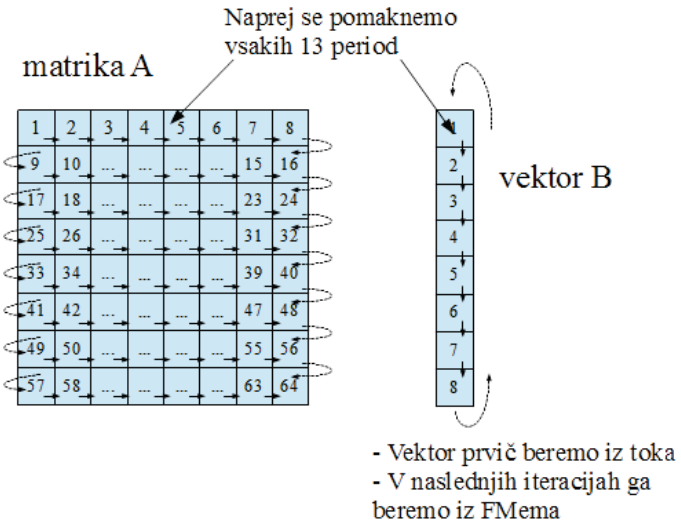
```
1   private static EngineInterface interfaceDefault() {
2     EngineInterface ei = new EngineInterface();
3
4     InterfaceParam matrixLength = ei.addParam("matrixLength",
          CPUTypes.INT);
5     InterfaceParam n = ei.addParam("n", CPUTypes.INT);
6     InterfaceParam loopLength =
          ei.getAutoLoopOffset("MatVecMultiplyKernel", "loopLength");
7     ei.ignoreAutoLoopOffset("MatVecMultiplyKernel", "loopLength");
8
9     ei.setTicks("MatVecMultiplyKernel", matrixLength*loopLength);
10
11    ei.setScalar("MatVecMultiplyKernel", "n", n);
12    ei.setStream("matInput", CPUTypes.FLOAT, matrixLength *
          CPUTypes.FLOAT.sizeInBytes());
13    ei.setStream("vecInput", CPUTypes.FLOAT,
          CPUTypes.FLOAT.sizeInBytes() * n);
14    ei.setStream("vecOutput", CPUTypes.FLOAT,
          CPUTypes.FLOAT.sizeInBytes() * n);
15
16    return ei;
17  }
18 }
```

Z objekti $InterfaceParam$ nastavimo vhodne parametre, ki jih bo lahko CPE posredovala nadzorniku. V zgornjem primeru jih bomo uporabili za izračun ustreznih dolžin tokov. V vrsticah 4 in 5 nastavimo vhodna parametra $matrixLength$, ki predstavlja velikost vhodne matrike, in $n$ za velikost vhodnega vektorja. Vrstici 6 in 7 poskrbita za čakajočo zanko, ki bo podrobneje opisana v naslednjem poglavju. V 9. vrstici nastavimo število urinih period izvajanja programa. V 11. vrstici nastavimo skalar. Gre za vrednost,

ki jo podamo iz CPE med samim izvajanjem in jo mora ščepec prebrati. V našem primeru sta to velikost vektorja in velikost matrike. Za razliko od parametrov, ki jih podamo ob inicializaciji ščepca in vplivajo na samo prevajanje kode, lahko skalarje spreminjamo iz CPE brez ponovnega prevajanja. V vrsticah od 12 do 14 nastavimo tri tokove: vhodno matriko, vhodni vektor in izhodni vektor (rezultat). Vsakemu pripada tudi tip podatkov (v tem primeru $float$) in dolžina toka v bajtih. V našem primeru bodo pri zagonu programa tokovi samodejno tudi parametri. Za narejeni vmesnik se iz datoteke C (ukazno-pretokovnega dela programa) kliče s klicem:

```
MatVecMultiply(size, n, mat, vec, output);
```

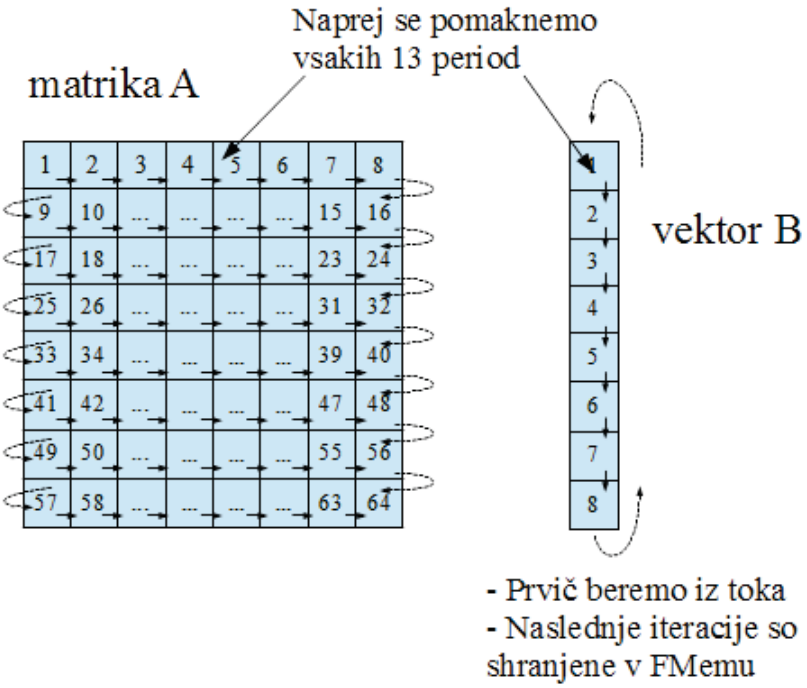
## 3.4 Osnovno množenje matrike in vektorja




Slika 3.1: Koreografija podatkov za osnovno množenje matrike in vektorja

Implementirajmo prvi ščepec. Zaenkrat bo cilj osnovna rešitev, ki se ne ozira na čas izvajanja. Na primeru bomo razložili, kako se na platformi programira, in v naslednjih poglavjih rešitev postopoma pohitrili.

V osnovni rešitvi bo koreografija podatkov v ščepcu podobna ukazno-pretokovni implementaciji; prikazuje jo slika 3.1. Elementi matrike bodo v program prihajali zaporedno po vrsticah. Tekom 1. vrstice matrike bodo sočasno prihajali elementi vektorja in se zraven shranjevali v hitri pomnilnik FMem. Pri naslednjih vrsticah matrike se bodo elementi vektorja brali iz FMema. Ob vsakem prihodu elementa matrike $A$ ga bomo množili z ustreznim elementom vektorja $B$.




Slika 3.2: Prikaz zanke za izračun vsote zmnožkov

Izračun vsote zmnožkov je malo bolj zapleten. Sešteti je treba zmnožke, ki niso na voljo ob istem času. To storimo tako, da implementiramo zanko, ki jo prikazuje slika 3.2. Računala bo delno vsoto zmnožkov in jo pošiljala nazaj, da bo dostopna ob prihodu novega zmnožka. Ob začetku vsake vrstice bo delna vsota enaka zmnožku enega elementa matrike in enega elementa vektorja. Zmnožek se bo poslal nazaj v graf, da bo dostopen ob prihodu naslednjega elementa matrike in vektorja. Ob koncu vsake vrstice bo delna

vsota vsebovala izračun enega elementa izhodnega vektorja in bo poslana na izhod. Sama operacija seštevanja traja 12 urinih period, zato bo delno vsoto treba poslati nazaj za 13 urinih period, hkrati pa kontrolirati vhod matrike in vektorja, da čaka 12 urinih period. Cevovod bo slabo izkoriščen, saj bo koristno porabljena samo vsaka 13. perioda in program zato ne bo učinkovit.

Poglejmo si izvorno kodo ščepca. V prvih desetih vrsticah bomo inicializirali sklop spremenljivk in iz ukazno-pretokovnega dela prebrali velikost matrike in vektorja $n$.

```
1  class MultiplyKernel extends Kernel {
2    final DFEType floatType = dfeFloat(8,24);
3
4    MultiplyKernel(KernelParameters parameters, int nMax) {
5      super(parameters);
6
7      Memory<DFEVar> ram = mem.alloc(floatType, nMax);
8      final int addrBits = MathUtils.bitsToAddress(nMax);
9      DFEType addrType = dfeUInt(addrBits);
10     DFEVar n = io.scalarInput("n", addrType);
```

Izven metode *main* začnemo z inicializacijo spremenljivke $floatType$ tipa $DFEType$, ki bo predstavljala obliko naših podatkov. Nastavimo ji tip podatka s plavajočo vejico in natančnostjo 32 bitov. S tem naredimo program bolj splošen, saj je za spremembo tipa treba spremeniti samo to vrstico. Konstruktor ščepca $MultiplyKernel$ sprejme (poleg standardnih parametrov) parameter $nMax$, ki predstavlja največjo dovoljeno velikost vektorja. V 8. vrstici s funkcijo $mem.alloc()$ dodelimo prostor za $nMax$ spremenljivk našega tipa. V 9. vrstici izračunamo število bitov vrednosti $nMax$ in na podlagi tega izdelamo nov tip. Naši števci bodo morali biti tega tipa, če jih bomo želeli uporabljati za dostop do pomnilnika (tako zahteva arhitektura Maxeler). V 11. vrstici preberemo skalar $n$, ki predstavlja velikost matrike in vektorja.

Sledi inicializacija števcev.

```
11     OffsetExpr loopLength = stream.makeOffsetAutoLoop("loopLength");
```

```
12      DFEVar loopLengthVal = loopLength.getDFEVar(this, dfeUInt(8));
13      CounterChain chain = control.count.makeCounterChain();
14      DFEVar i = chain.addCounter(n, 1);
15      DFEVar j = chain.addCounter(n, 1);
16      DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);
```

Podatkovna-pretokovna arhitektura ne pozna zank v klasičnem smislu (zanke for ali while). Maxeler ima na voljo števce, s katerimi kontroliramo, kdaj se bo prebral naslednji element toka. V vrstici 12 uporabimo funkcijo MaxCompilerja, ki na podlagi notranjih zank v kodi (zank, ki pošiljajo podatke nazaj) samodejno izračuna najmanjšo možno vrednost odmika, in jo v 13. vrstici spremenimo v obliko, kompatibilno s števci. V vrsticah od 14 do 17 inicializiramo verigo števcev. Najnižji števec $loopCounter$ šteje od 0 do $loopLengthVal$ (v našem primeru 12) in predstavlja našo notranjo zanko, ki čaka na seštevek dveh števil. Po koncu se postavi nazaj na nič, $j$ pa se poveča za 1. $j$ teče od 0 do $n-1$ in predstavlja stolpec trenutnega elementa matrike. Podobno nam $i$ pove vrstico trenutnega elementa.

Tokove beremo na naslednji način:

```
17      DFEVar readVec = i === 0 & loopCounter === 0;
18      DFEVar vecInput = io.input("vecInput", dfeFloat(8,24), readVec);
19      ram.write(j, vecInput, readVec);
20      DFEVar vec = readVec ? vecInput : ram.read(j);
21      DFEVar mat = io.input("matInput", floatType, loopCounter ===
            (loopLengthVal-1));
```

V vrstici 18 inicializiramo spremenljivko $readVec$, ki bo $True$ za vrednosti, ko želimo prebrati element vektorja in ga zapisati v pomnilnik. To bo vsakih $loopLengthVal$ urinih period ($loopCounter = 0$) za skupaj $n$ branj (dokler $i = 0$). Potem bomo element vektorja brali direktno iz pomnilnika. V vrstici 19 preberemo element vhodnega vektorja in ga v 20. vrstici zapišemo v pomnilnik (če je spremenljivka $readVec = True$) na mesto $j$ ($j$ predstavlja tudi stolpec trenutnega elementa matrike). V 21. vrstici nastavimo trenutni element vektorja na trenutno prebranega za prvo vrstico matrike ali ga preberemo iz pomnilnika za vse ostale vrstice matrike (to je treba narediti, ker

zapis potrebuje eno urino periodo, da se zapiše v pomnilnik).

V naslednjih štirih vrsticah bomo za izračun vsote izdelali notranjo zanko.

```
22    DFEVar carriedSum = floatType.newInstance(this);
23    DFEVar sum = j === 0 ? 0.0 : carriedSum;
24    DFEVar newSum = mat * vec + sum;
25    carriedSum <== stream.offset(newSum, -loopLength);
```

V 23. vrstici spremenljivki $carriedSum$ z $newInstance(this)$ nastavimo tok, ki je zaenkrat še brez vhoda. Sama zanka je sestavljena iz treh delov. Najprej ji v 24. vrstici nastavimo vrednost 0, če gre za začetek vrstice (trenutni element matrike je iz 1. stolpca) ali delno vsoto ob naslednjih iteracijah. V 25. vrstici izvedemo izračun: delni vsoti prištejemo zmnožek trenutnega elementa matrike in vektorja. V 26. vrstici z <== nastavimo vhod spremenljivki $carriedSum$. Ta bo enaka $stream.offset(newSum, -loopLength)$ in bo enaka spremenljivki $newSum$ pred $loopLength$ urinimi periodami.

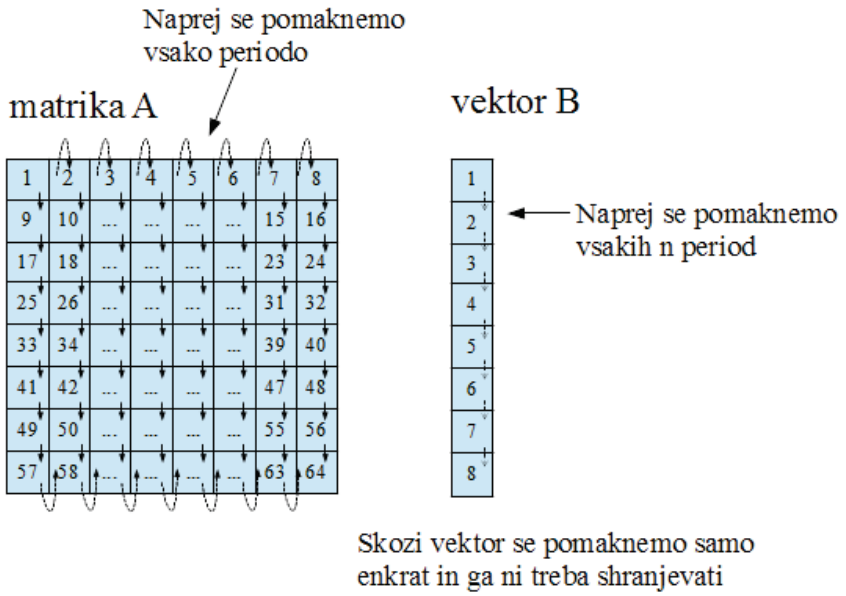
Poskrbeti je treba še za izhodni tok.

```
26    io.output("vecOutput", newSum, floatType, j === (n - 1) & loopCounter
          === (loopLengthVal-1));
27    }
28 }
```

Na koncu vsake vrstice (stolpec = $n$-1 in $loopCounter$ je na najvišji vrednosti) pošljemo delno vsoto ($carriedSum$) na izhod. S tem je program končan.

## 3.5 Transponiranje matrike

Naslednji izziv bo odstranitev notranje zanke. Zaradi čakanja na podatke (na delno vsoto) pri izračunu vsote zelo upočasnimo izvajanje programa. Pri izračunu vsote za isti element $c_i$ vektorja $C$ so podatki med seboj odvisni in mora biti delni seštevek končan, da se lahko program nadaljuje. Za različni $c_i$ pa to ni potrebno. Medtem ko čakamo na seštevek delne vsote $c_1$, bi lahko izkoristili čas in začeli izračunavati delne vsote naslednjih $c_2$ do $c_n$. S transponiranjem matrike lahko dosežemo, da podatki v ščepec ne bodo




Slika 3.3: Koreografija podatkov za algoritem s transponiranjem matrike

več tekli po vrsticah, ampak po stolpcih. Vsota se tako ne bo delala med dvema zaporednima zmnožkoma, ampak nad vsakim $n$-tim zmnožkom. Novo koreografijo podatkov prikazuje slika 3.3. V ščepcu namesto čakanja na delni izračun vsote takoj nadaljujemo z elementom, ki je v istem stolpcu v naslednji vrstici. Ker se premikamo po stolpcih, bomo prvih $n$ elementov zmnožkov množili z elementom $b_1$. V naslednjih $n$ korakih bomo množili z elementom $b_2$ in tako naprej do $b_n$. Zaradi tega vektorja $b$ ni več treba shranjevati v FMem, ampak je dovolj, da ga beremo v pravem trenutku. Element bo ostal v grafu, dokler ne sprožimo novega branja vektorja (naslednjih $n$ iteracij). Vsako delno vsoto tokrat odmaknemo nazaj za $n$. S tem nam bo na voljo ravno ob prihodu v nov stolpec matrike.

Ker ne potrebujemo več pomnilnika in števca za čakanje, se ščepec poenostavi:

```
1 MultiplyKernel(KernelParameters parameters, int nMax) {
2   super(parameters);
3   DFEVar n = io.scalarInput("n", dfeUInt(32));
```

```
4     CounterChain chain = control.count.makeCounterChain();
5     DFEVar i = chain.addCounter(n, 1);
6     DFEVar j = chain.addCounter(n, 1);
7
8     DFEVar vec = io.input("vecInput", dfeFloat(8,24), j===0);
9     DFEVar mat = io.input("matInput", floatType);
10
11    DFEVar carriedSum = floatType.newInstance(this);
12    DFEVar sum = i === 0 ? 0.0 : carriedSum;
13    DFEVar newSum = mat*vec+sum;
14    carriedSum <== stream.offset(newSum, n.cast(dfeInt(32))*-1, -nMax,
          -16);
15
16    io.output("output", newSum, floatType, i === (n-1));
17 }
```

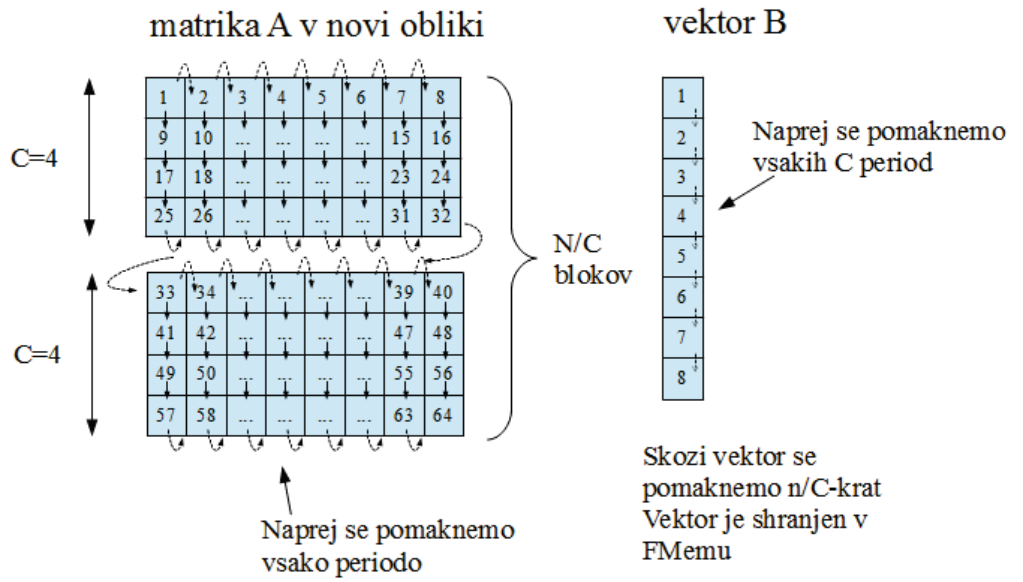
V 3. vrstici lahko sedaj skalar beremo v obliki $dfeUint(32)$, ker ga ne potrebujemo več za naslavljanje pomnilnika. Nova veriga števcev vsebuje samo spremenljivki $i$ in $j$. Element matrike bomo brali vsako urino periodo, element vektorja pa vsako $n$-to periodo. Na izhod bomo izračunane elemente poslali v zadnjih $n$ periodah programa.

V nadzorniku je edina sprememba število urinih period izvajanja programa, ki je zdaj enako dolžini matrike.

```
ei.setTicks("MatVecMultiplyKernel", matrixLength);
```

## 3.6 Zmanjšanje odmika

Odmik velikosti $n$ iz prejšne različice algoritma nam za velike matrike po nepotrebnem porablja sredstva čipša. Smiselno bi bilo, da uporabljamo odmik, ki je enak času, ki ga porabi seštevalnik za seštevanje dveh števil. S tem maksimalno izkoristimo cevovod in hkrati ne porabimo veliko sredstev. Temu pravijo tudi C-slow retiming [32]. Da to dosežemo, moramo drugače razporediti vhod. Slika 3.4 prikazuje koreografijo podatkov v novem algoritmu. Za lažjo razlago smo jo poenostavili za primer, kjer bi bil odmik 4.




Slika 3.4: Koreografija podatkov za algoritem z zmanjšanjem odmika in velikostjo odmika 4

Ponovno bomo uporabljali pomnilnik vezja, zato bo ščepec bolj podoben osnovni implementaciji.

```
1 MultiplyKernel(KernelParameters parameters, int nMax, int C) {
2   super(parameters);
3
4   final int addrBits = MathUtils.bitsToAddress(nMax);
5   DFEType addrType = dfeUInt(addrBits);
6   DFEVar n = io.scalarInput("n", addrType);
```

Da naredimo ščepec bolj univerzalen (npr. če bi se spremenile operacije seštevanja in množenja), dodamo v konstruktorju spremenljivko $C$, ki predstavlja minimalen potreben odmik. Tudi če je malo večji, bistveno ne vpliva na program. Podatki se bodo še vedno prenašali vsako urino periodo in cevovod bo maksimalno izkoriščen.

```
1   CounterChain chain = control.count.makeCounterChain();
2   DFEVar b = chain.addCounter(n/C, 1);
3   DFEVar i = chain.addCounter(n, 1);
```

```
4     DFEVar j = chain.addCounter(C, 1);
5
6     DFEVar vec = io.input("vecInput", dfeFloat(8,24), j===0 & b === 0);
7     DFEVar input = io.input("matInput", floatType);
```

Spremenimo tudi verigo števcev. Spremenljivka $b$ predstavlja, v katerem bloku matrike se nahajamo, spremenljivka $i$ predstavlja vrstico bloka in spremenljivka $j$ stolpec. Vektor iz vhoda beremo samo enkrat, sočasno z elementom matrike iz prve vrstice prvega bloka. Zatem ga beremo iz FMema.

```
1     Memory<DFEVar> ram = mem.alloc(floatType, nMax);
2     ram.write(i, vec, j===0 & b === 0);
3     DFEVar elFromRam = j===0 & b === 0 ? vec : ram.read(i);
4
5     DFEVar carriedSum = floatType.newInstance(this);
6     DFEVar sum = i === 0 ? 0.0 : carriedSum;
7     DFEVar newSum = input*elFromRam+sum;
8
9     carriedSum <== stream.offset(newSum, -C);
10    io.output("output", newSum, floatType, i === (n-1));
11 }
```

Za pravočasno branje in pisanje poskrbimo s števci, podobno kot v prejšnjih primerih. Odmik je tokrat enak spremenljivki $C$. Na izhod pošiljamo, ko se nahajamo v zadnjem stolpcu.

Matriko spremenimo v pravo obliko v ukazno-pretokovnem delu z naslednjo funkcijo:

```
1  void transform_C(int n, float *inputMatrix, float *matTransformed, int C){
2     int count = 0;
3     for (int b = 0; b < n; b += C) {
4        for ( int x = 0; x < n; ++x) {
5           for ( int y = b; y < b + C; ++y) {
6              matTransformed[count] = inputMatrix[y * n + x];
7              count++;
8           }
9        }
10    }
```

```
11 }
```

V nadzorniku dodamo na vrhu statično spremenljivko.

```
static final int C = 16;
```

in spremenljivko podamo konstruktorju.

```
Kernel kernel = new MultiplyKernel(manager.makeKernelParameters(), nMax,
    C);
```

## 3.7 Paralelizacija z uporabo cevi

Poleg izkoriščanja cevovoda, ki je pri matričnem množenju zaradi preprosti relativno majhen, lahko algoritem dodatno paraleliziramo s tako imenovanimi cevmi (angl. pipes). To storimo tako, da na podatkovno-pretokovno vezje povežemo več vzporednih vhodnih in izhodnih tokov ter celoten program na vezju fizično pomnožimo. Tega ne moremo narediti za poljubne velikosti. Omejeni smo s prepustnostjo med ukazno-pretokovnim in podatkovno-pretokovnim delom ter samo velikostjo vezja.

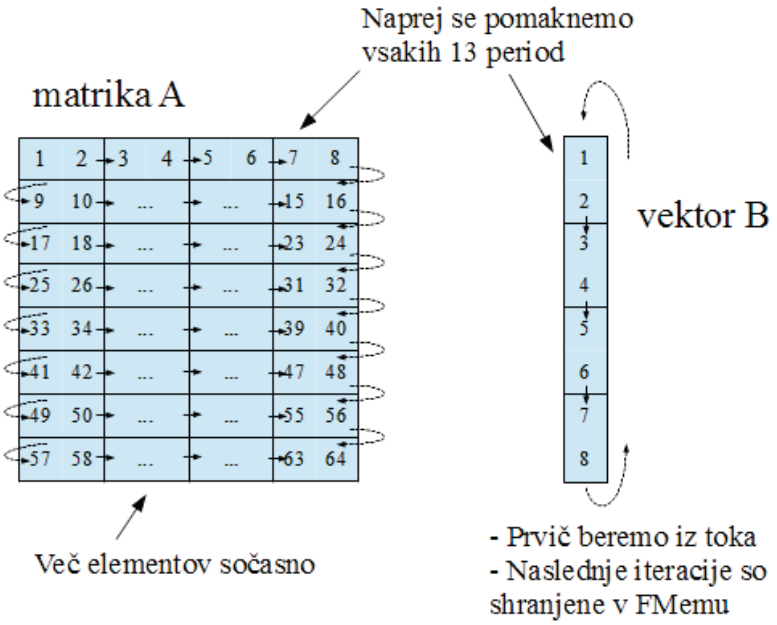
Možnih koreografij podatkov pri uporabi cevi je več. Bistvenih razlik pri hitrosti tu nismo pričakovali, zato smo za vsak algoritem implementirali tisto, ki se nam je zdela najbolj intuitivna nadgradnja različic brez cevi. Koreografijo za osnovno verzijo množenja za primer dveh cevi prikazuje slika 3.5. Množimo dva zaporedna elementa matrike v prvotni obliki z dvema zaporednima elementoma iz vektorja. Cevi vsebujejo delne vsote za isti element izhodnega vektorja, zato jih je treba na koncu vrstice sešteti.

Za vpeljavo cevi v osnovno verzijo programa iz razdelka 3.4 je treba narediti naslednje. Najprej spremenimo konstruktor tako, da mu dodamo spremenljivko *vectorDepth*, ki nam bo povedala število cevi.

```
MultiplyKernel(KernelParameters parameters, int nMax, int vectorDepth)
```

Znotraj konstruktorja definiramo nov tip *vectorType*

```
final DFEVectorType<DFEVar> vectorType = new
    DFEVectorType<DFEVar>(floatType, vectorDepth);,
```




Slika 3.5: Paralelizacija osnovne verzije v primeru dveh cevi

ki predstavlja skupek elementov tipa $floatType$. Tako zmnožki kot delne vsote so med seboj neodvisni in se lahko izvajajo paralelno.

V ščepcu algoritem posodobimo tako, da naše vhode, izhode in vmesne manipulacije podatkov obdamo z elementom $DFEVector$. Na ta način se vse naše operacije branja tokov, množenja in seštevanja na vezju fizično pomnožijo. Popraviti je treba števce:

```
CounterChain chain = control.count.makeCounterChain();
DFEVar i = chain.addCounter(n, 1);
DFEVar j = chain.addCounter(n, vectorDepth);
DFEVar loopCounter = chain.addCounter(loopLengthVal, 1);
```

Števec $j$, ki predstavlja stolpec trenutnega elementa matrike, se bo sedaj povečeval za velikost vektorja $vectorDepth$. Za vsoto vektorja uporabimo funkcijo $TreeReduce.reduce$ iz knjižnice maxpower [15].

```
DFEVar[] summands = new DFEVar[vectorDepth];

DFEVector<DFEVar> elFromRam = vectorType.newInstance(this);
```

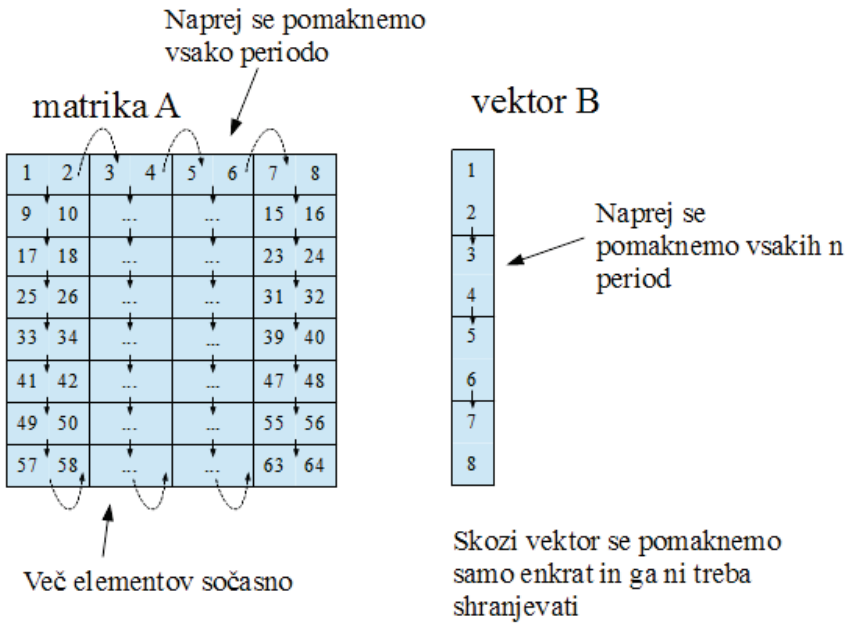
```
    elFromRam = i === 0 ? vec : ram.read(j);

    newSum <== mat * elFromRam + sum;
    carriedSum <== stream.offset(newSum, -loopLength);

    for(int s=0; s < vectorDepth; s++){
       summands[s] = newSum[s];
    }
    io.output("output", TreeReduce.reduce(new Add<DFEVar>(), summands),
        floatType, j ===
    (n-vectorDepth) & loopCounter === (loopLengthVal-1));
}
```

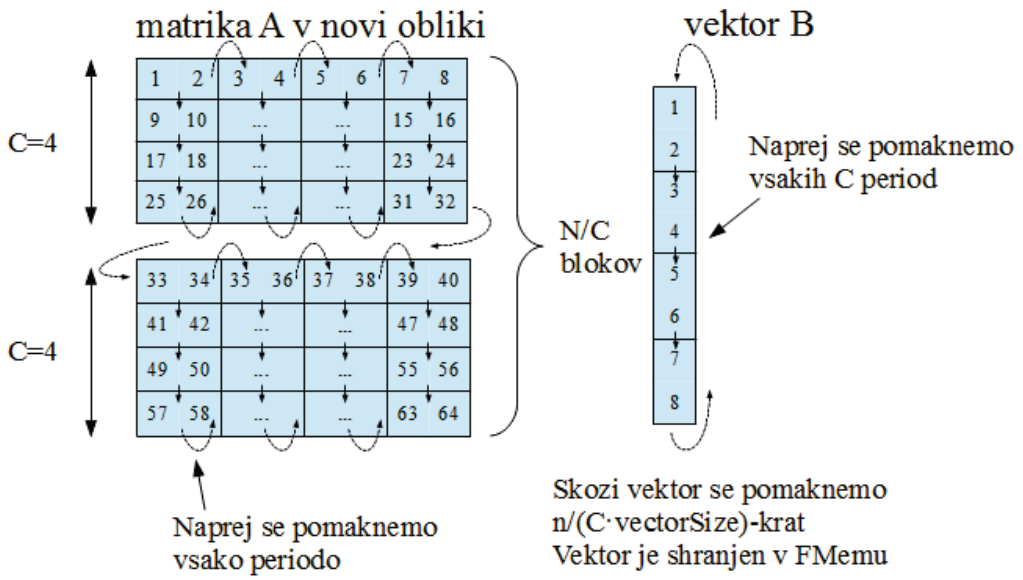
V nadzorniku bomo število urinih period izvajanja delili z velikostjo vektorja.

```
ei.setTicks("MatVecMultiplyKernel", matrixLength*loopLength/vectorSize);
```




Slika 3.6: Paralelizacija transponirane verzije v primeru dveh cevi

Sliki 3.6 in 3.7 prikazujeta koreografije podatkov za druga dva algoritma. Pri osnovni verziji lahko matrika ostane v izvorni obliki. Pri algoritmu s tran-




Slika 3.7: Paralelizacija verzije z zmanjšanjem odmika v primeru dveh cevi

sponiranjem matrike razporedimo v ukazno-pretokovnem delu na naslednji način:

```
void transform (int n, float *input, float *matrTrans, int vecSize){
   for (int v = 0; v < n; v=v+vecSize) {
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < vecSize; j++) {
            matTrans[i*vecSize +j + v*n] = input[j+i*n+v];
         }
      }
   }
}
```

Matriko pri algoritmu z zmanjšanjem vrstice spremenimo z:

```
void transform_vec_C(int n, float *mat, float *matrixTransformed, int
    vecSize, int C){
   int count = 0;
      for (int jj = 0; jj < n; jj += C) {
         for ( int i = 0; i < n/vecSize; i++) {
```

```
            for ( int j = jj; j < jj + C; j++) {
               for (int v = 0; v < vecSize; v++){
                  matrixTransformed[count] = mat[j * n + i*vecSize+v];
                  count++;
            }
         }
      }
   }
}
```

## 3.8 Množenje z naborom vektorjev

Lotimo se še problema množenja z naborom vektorjev. Namesto da matriko množimo z enim vektorjem, jo množimo z več vektorji. Pri opisu podatkovno-pretokovnega računanja v razdelku 2.5 je bila ponovna uporaba podatkov ena izmed smernic za dosego občutnih pohitritev. Zaradi narave problema je bilo to mogoče narediti pri vektorju, ne pa pri matriki, ki predstavlja večino vhodnih podatkov. V primeru množenja matrike z več vektorji bi bila ta večkrat uporabljena. Problem je zanimiv tudi iz praktičnega vidika. Matrika pogosto predstavlja transformacijo, ki jo je treba izračunati za več različnih vhodnih vektorjev.

Za osnovo smo vzeli algoritem z zmanjšanjem odmika. Kot smo omenili, bodo zdaj poleg elementov vektorja večkrat uporabljeni tudi elementi matrike in bi bilo zato smiselno narediti večkratni dostop čim hitrejši. Matrika je prevelika, da bi jo lahko shranili v hitri pomnilnik FMem. Na voljo nam je veliki pomnilnik LMem. Sicer je občutno počasnejši od FMema in ne omogoča naključnega dostopa, vendar bomo vanj lahko shranili celotno matriko. Iz LMema jo bomo brali kot vhodni tok. LMem bo cevi lahko bolj izkoriščal, zato bomo med eksperimentalnim ovrednotenjem uporabili 48 cevi. Bolj podrobno bomo različno število cevi analizirali pri matričnem množenju v naslednjem poglavju.

Samega ščepca nam ni treba spreminjati. Prilagoditi moramo samo nad-

zornik in ukazno-pretokovni del. LMem deluje v skupkih po 384 bajtov. V našem primeru (kjer element matrike zavzame 4 bajte) to pomeni, da mora biti število elementov deljivo s 96. Problem rešimo tako, da v ukazno-pretokovnem delu matriki dodamo ustrezne ničle. Matriko in parametre shranimo iz ukazno-pretokovnega dela v LMem na naslednji način:

```
max_file_t * maxfile = MatMatMultiply_init();
max_engine_t * engine = max_load(maxfile, "*");

MatMatMultiply_writeLMem_actions_t writeact;
writeact.param_address = 0;
writeact.param_nbytes = data_size_bytes1;
writeact.instream_cpu_to_lmem = mat;
MatMatMultiply_writeLMem_run(engine, &writeact);

MatMatMultiply_actions_t actions;
actions.param_matrixLength = n1*n1;
actions.param_n = n1;
```

Najprej nastavimo naslov za pisanje in dolžino podatkov. Bistvena je funkcija *MatMatMultiply_writeLMem_run*, ki shrani matriko v LMem. Za množenje z naborom vektorjev kličemo z zanko *for* ščepec za različen vhodni vektor:

```
for (int i=0; i<st_vec; i++) {
   actions.instream_vectorInput = &vektorji_in[i*n];
   actions.outstream_output = &vektorji_out[i*n];
   MatMatMultiply_run(engine, &actions);
}

max_unload(engine);
```

V nadzorniku v metodi *main* posodobimo tipe vhodov. Matrika bo prišla iz LMema; vhodni in izhodni vektor bosta prišla iz CPE.

```
manager.setIO(
link("matrixInput", IODestination.LMEM_LINEAR_1D),
link("vectorInput", IODestination.CPU),
link("output", IODestination.CPU)
```

```
);
```

V vmesniku $interfaceDefault$

```
EngineInterface ei = new EngineInterface();
InterfaceParam zero = ei.addConstant(0l);
ei.setLMemLinear("matrixInput", zero, matrixSizeInBytes);
```

spremenimo funkcijo $setStream()$ v $setLMemLinear()$, ki bo matriko iz LMema brala zaporedoma.

## 3.9 Eksperimentalno ovrednotenje

### 3.9.1 Postavitev eksperimenta

Eksperimentalno ovrednotenje opisanih algoritmov smo izvedli na računalniku, ki vsebuje ukazno-pretokovne in podatkovno-pretokovne enote ter operacijski sistem MaxelerOS 2016.1.

- Podatkovni-pretokovni del je kartica Maxeler Vectis MAX3424A PCI-express kartica. Na kartici je vezje FPGA Xilinx Virtex 6 SXT475, ki ima 297600 logičnih enot (LUT), $2 \times 297600$ flip-flopov, 2016 signalnih procesorjev (DSPjev) in 18 kBitov BRAMa. Zraven je tudi veliki pomnilnik LMem v obliki 4 DIMM enot z velikostjo 16 GB, s skupno velikostjo 64 GB in prenosom 2133 MT/s. Vectis kartica je v reži PCI-express 3.0 s pasovno širino 16.000 MB/s.

- Ukazno-pretokovni del vsebuje CPE Intel(R) Core(TM) i7-6700K CPE, ki teče s frekvenco 4 GHz in ima 3 stopnje predpomnilnikov (L1 cache: 32K+32K, L2 cache: 256K, L3 cache: 8192K).

Na istem računalniku smo testirali tudi ukazno-pretokovno implemenetacijo. Testirane algoritme in povzetke delovanj prikazuje tabela 3.1. Testna množica so za problem množenja matrike in vektorja kvadratne matrike velikosti od $1024 \times 1024$ do $16384 \times 16384$, ki jo povečujemo v korakih po 1024.

| **Ime algoritma** | **Povzetek delovanja** |
|---|---|
| CPU | Ukazno-pretokovni algoritem za množenje matrike in vektorja, implementiran v programskem jeziku C. |
| Basic | Osnovno množenje matrike in vektorja. Podatkovno-pretokovni algoritem, kjer je koreografija pretakanja podatkov enaka kot v ukazno-pretokovni implementaciji in je vmes treba čakati na delno vsoto. |
| Transposed | Algoritem s transponiranjem matrike. Vhodno matriko smo transponirali in računali po stolpcih, zaradi česar se znebimo čakanja na delno vsoto. |
| Tiled | Algoritem z zmanjšanjem odmika. Vhodno matriko smo prerazporedili na način, da se znebimo čakanja na delne vsote in zmanjšamo odmik zanke. |
| <Algoritem>Vec<št.> | Implementacija s cevmi za izbrani algoritem in izbrano število cevi. |
| multivec_CPU | Ukazno-pretokovni algoritem za množenje matrike z naborom vektorjev. |
| MatVecMultiLMem | Podatkovno-pretokovni algoritem za množenje matrike z naborom vektorjev, ki uporablja LMem in 48 cevi. |

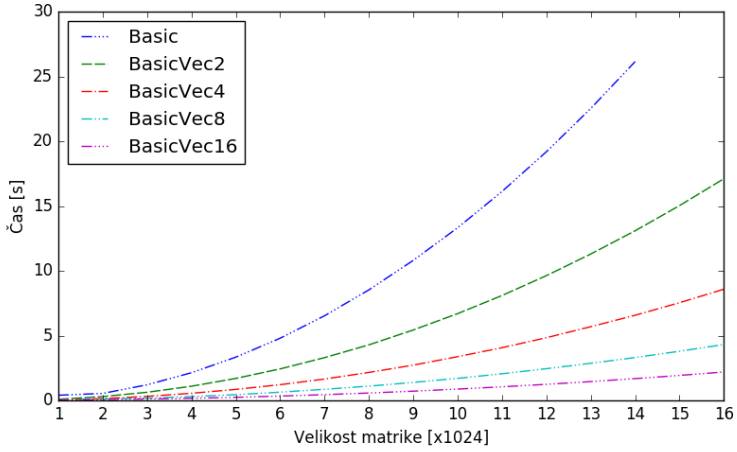
Tabela 3.1: Pregled testiranih algoritmov skupaj s povzetki delovanja

Velikost vektorja je enaka velikosti ene vrstice matrike. Matrika in vektor sta generirana naključno z elementi velikosti od $-200$ do 200.

Programi so se tudi za največje matrike izvedli v nekaj sekundah, zato smo za boljšo oceno časa izvajanja vzeli povprečje petih vrednosti. Testna

množica za problem množenja z več vektorji je vsebovala matriko velikosti $3072 \times 3072$ in vektorje velikosti 3072. Število vektorjev za množenje je bilo od 1024 do 20480 v korakih po 1042.

### 3.9.2 Rezultati

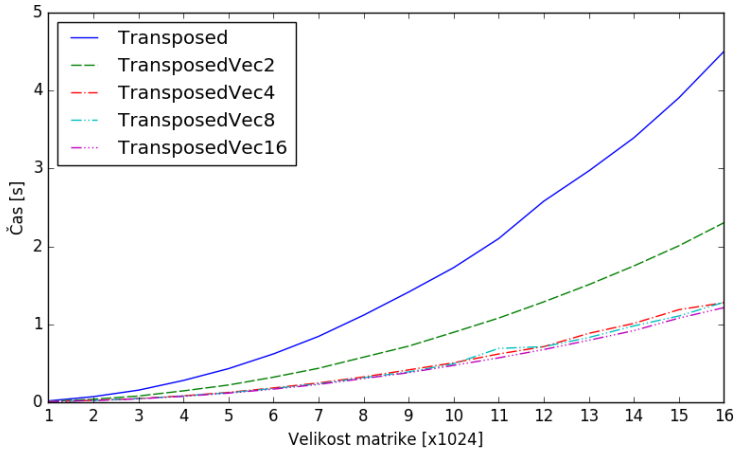



Slika 3.8: Primerjava časa izvajanja za osnovne algoritme množenja matrike z vektorjem pri različnem številu cevi

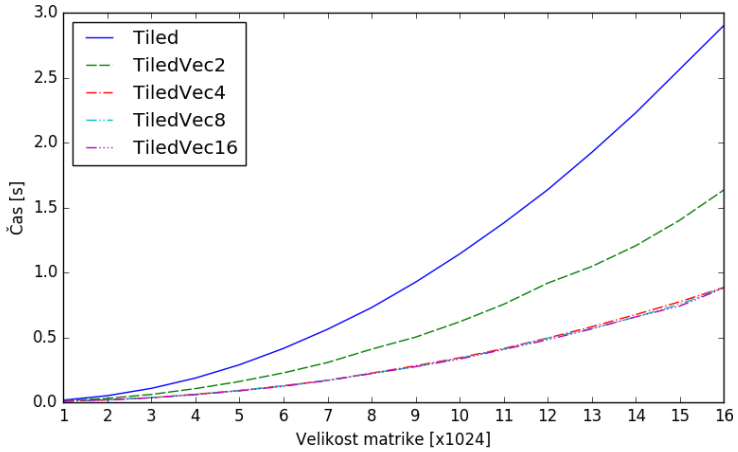
Poglejmo si rezultate eksperimentalnega ovrednotenja. Slika 3.8 prikazuje, kako so cevi izboljšale osnovni algoritem. Občutne izboljšave dosegamo vse do šestnajstih cevi. Za več cevi je bilo premalo sredstev čipa in programa nismo mogli prevesti. Pri verziji brez cevi se je osnovna verzija zaradi neznanega razloga za zadnji dve vrednosti ustavila in zato manjkata na grafu.

Sliki 3.9 in 3.10 prikazujeta, kako sta se z uporabo cevi pohitrila algoritem s transponiranjem in algoritem z zmanjšanjem odmika. V obeh primerih so cevi na hitrost vplivale vse do štirih cevi. Več cevi na čas izvajanja ni več vplivalo. Podobne rezultate so dobili v [3]. Gre za ozko grlo med ukazno-pretokovnim in podatkovno-pretokovnim delom računalnika.

Primerjavo časa izvajanja podatkovno-pretokovnih algoritmov z ukazno-pretokovnim prikazuje slika 3.11. Izkaže se, da je tudi najhitrejši podatkovno-pretokovni algoritem občutno počasnejši od ukazno-pretokovnega. Slika 3.12

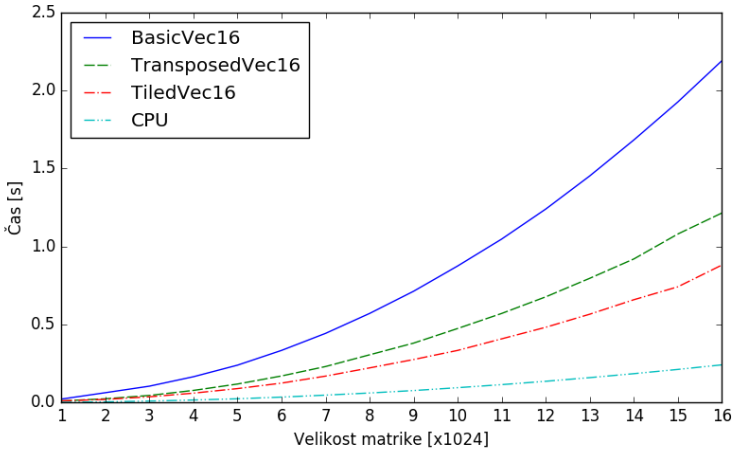



Slika 3.9: Primerjava časa izvajanja za algoritme s transponiranjem pri različnem številu cevi

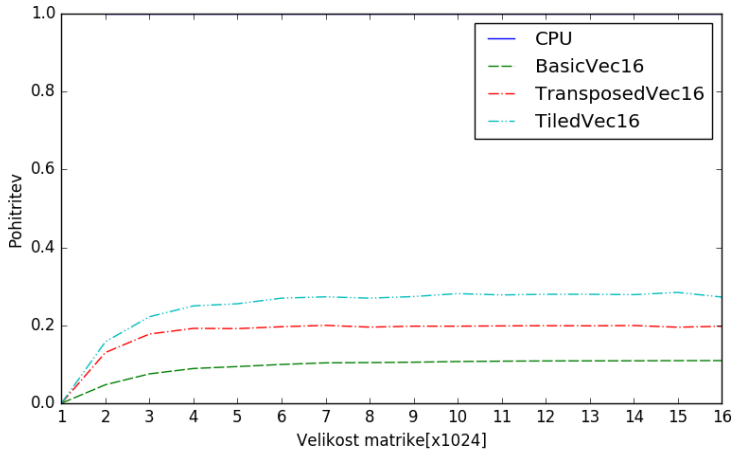



Slika 3.10: Primerjava časa izvajanja zmanjšanjem odmika pri različnem številu cevi

prikazuje faktor pohitritve podatkovno-pretokovnih algoritmov v primerjavi z ukazno-pretokovnim. Pri najhitrejšem podatkovno-pretokovnem algoritmu je faktor pohitritve manjši od 0,3, kar pomeni, da je algoritem skoraj 4-krat počasnejši od ukazno-pretokovnega algoritma. Na prvi pogled se to zdi slabo, ampak v bistvu pohitritve sploh nismo pričakovali. Vnaprej smo vedeli, da so slabe možnosti za izkoriščanje dolgega cevovoda ali ponovne



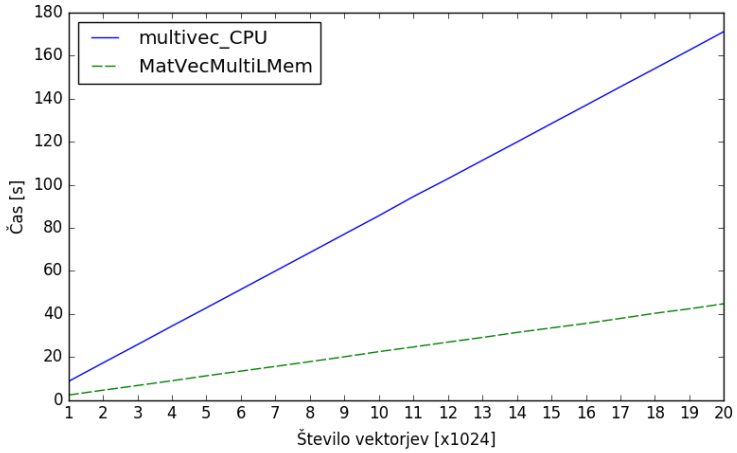

Slika 3.11: Primerjava časa izvajanja najhitrejših različic algoritmov skupaj z ukazno-pretokovnim algoritmom



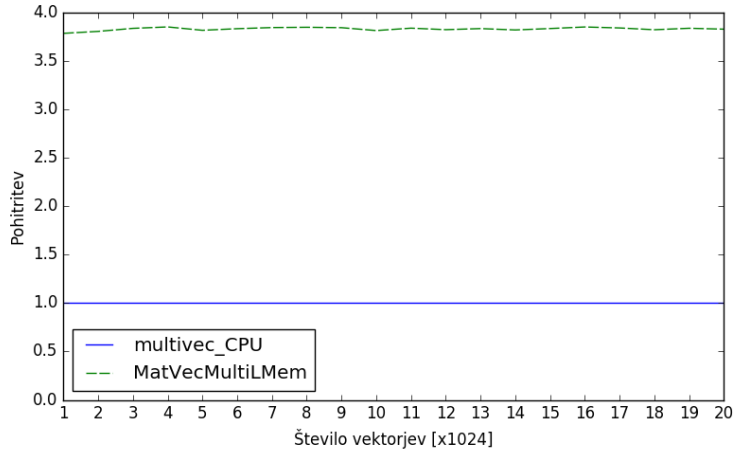

Slika 3.12: Faktor pohitritve najhitrejših algoritmov v primerjavi z ukazno-pretokovno različico

uporabe podatkov. Omejuje nas tudi prepustnost. Na vezju imamo prostora za več cevi, kot smo jih uporabljali, in bi v primeru večje prepustnosti lahko čase izvajanja občutno izboljšali.

Algoritem množenja z naborom vektorjev se je odrezal veliko bolje. Uporabili smo matriko velikosti $3072 \times 3072$ in testirali za število vektorjev od 1024 do 20480. Na slikah 3.13 in 3.14 vidimo, da so pohitritve skoraj 4-




Slika 3.13: Primerjava časa izvajanja množenja matrike velikosti $3072 \times 3072$ z naborom vektorjev




Slika 3.14: Faktor pohitritve množenja matrike velikosti $3072 \times 3072$ z naborom vektorjev

kratne. Že pri 1024 vektorjih je bila dosežena občutna pohitritev in dodatno število vektorjev ni več občutno vplivalo na pohitritev. Ponovna uporaba podatkov matrike je bila torej v našem primeru nujna za doseg pohitritev v primerjavi z ukazno-pretokovno različico.

### 3.9.3 Poraba sredstev vezja FPGA

Poglejmo si porabo sredstev za različne algoritme. Deleže porabe v procentih prikazujejo tabele 3.2, 3.3 in 3.4. Pri vseh algoritmih je ostal prostor za dodatne operacije. To nakazuje, da bi pri sorodnem problemu s kompleksnejšim vmesnim izračunom (namesto operacije množenja) lahko dosegli boljše pohitritve v primerjavi z ukazno-pretokvnim algoritmom. Pri algoritmu z zmanjšanjem odmika je v primerjavi z algoritmom s transponiranjem nekoliko manjša poraba. Omenili smo že, da bi boljša prepustnost med ukazno-pretokovnim in podatkovno-pretokovnim delom izboljšala rezultate. Zaradi dovolj prostega prostora bi bilo algoritme mogoče prevesti z večjim številom cevi.

| Algoritem | LUT | FF1 | FF2 | BRAM | DSP |
|---|---|---|---|---|---|
| Basic | 2,3 | 2,14 | 0,56 | 3,2 | 0,1 |
| BasicVec2 | 2,3 | 2,14 | 0,56 | 3,2 | 0,1 |
| BasicVec4 | 3,4 | 3,4 | 0,81 | 8,18 | 0,4 |
| BasicVec8 | 4,93 | 4,68 | 1,38 | 15,79 | 0,79 |
| BasicVec16 | 10,41 | 8,69 | 1,32 | 30,17 | 1,59 |

Tabela 3.2: Poraba sredstev vezja FPGA osnovnega algoritma pri različnem številu cevi

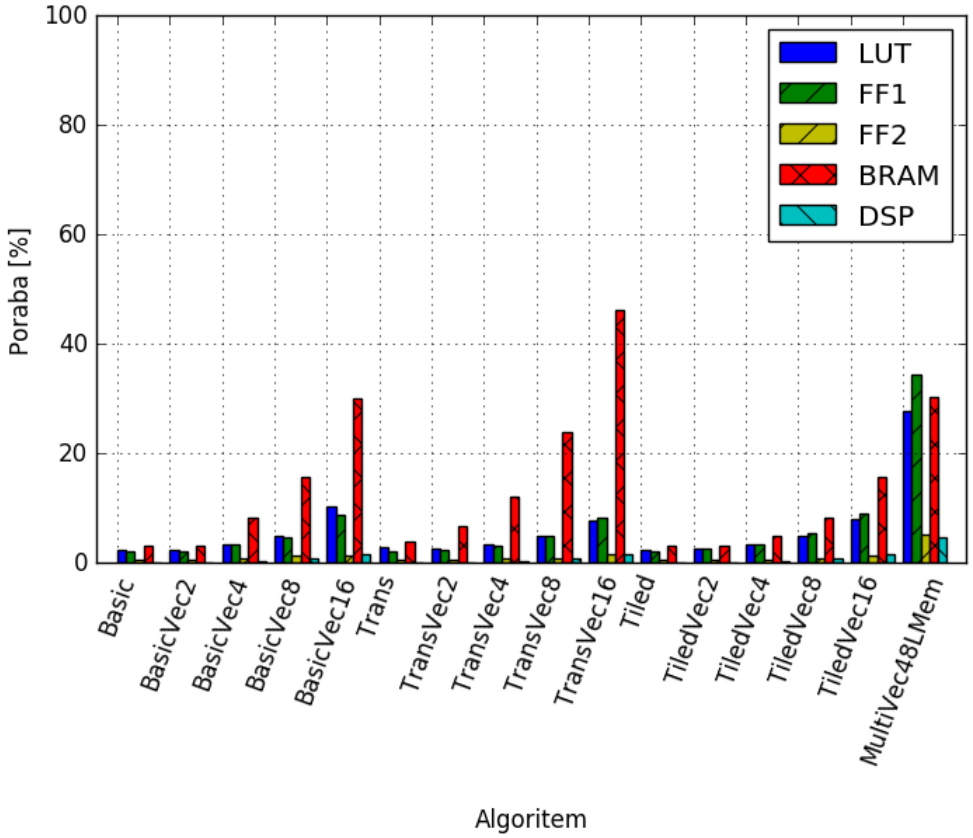
Na sliki 3.15 je lepo razvidna primerjava porabe sredstev vseh testiranih algoritmov. Tudi algoritem množenja z naborom vektorjev (zadnji na sliki) smo brez težav prevedli z 48 cevmi

| Algoritem | LUT | FF1 | FF2 | BRAM | DSP |
|---|---|---|---|---|---|
| Trans | 2,83 | 2,17 | 0,58 | 4,04 | 0,1 |
| TransVec2 | 2,75 | 2,52 | 0,72 | 6,86 | 0,2 |
| TransVec4 | 3,42 | 3,26 | 0,81 | 12,17 | 0,4 |
| TransVec8 | 4,98 | 4,88 | 0,79 | 23,83 | 0,79 |
| TransVec16 | 7,9 | 8,36 | 1,55 | 46,29 | 1,59 |

Tabela 3.3: Poraba sredstev vezja FPGA algoritma s transponiranjem pri različnem številu cevi

| Algoritem | LUT | FF1 | FF2 | BRAM | DSP |
|---|---|---|---|---|---|
| Tiled | 2,35 | 2,08 | 0,63 | 3,2 | 0,1 |
| TiledVec2 | 2,69 | 2,66 | 0,58 | 3,2 | 0,2 |
| TiledVec4 | 3,36 | 3,53 | 0,6 | 4,98 | 0,4 |
| TiledVec8 | 4,99 | 5,38 | 0,86 | 8,18 | 0,79 |
| TiledVec16 | 8,15 | 8,96 | 1,4 | 15,79 | 1,59 |
| MultiVec48LMem | 27,84 | 34,47 | 5,18 | 30,26 | 4,76 |

Tabela 3.4: Poraba sredstev vezja FPGA algoritma z zmanjšanjem odmika pri različnem številu cevi in algoritma množenja z naborom vektorjev




Slika 3.15: Primerjava porabe sredstev vezja FPGA vseh implementiranih algoritmov

# Poglavje 4

# Matrično množenje

Matrično množenje je tako kot množenje matrike in vektorja ena ključnih operacij v računalništvu. Uporablja se pri delu z grafiko, v robotiki ter pri obdelavi slik in procesiranju signalov. Veliko dela je bilo že vloženega v zmanjšanje asimptotične časovne zahtevnosti algoritmov (npr. [30], [10]). Gre za teoretične algoritme z dobro časovno asimptotično zahtevnostjo. Nas bo zanimal predvsem čas izvajanja v praksi. V poglavju bomo implementirali več različic matričnega množenja za podatkovno-pretokovni računalnik. Za osnovo bomo vzeli algoritme iz poglavja 3 in jih prilagodili za matrično množenje. Uporabljali bomo veliki pomnilnik LMem in večje število cevi. Na koncu bomo implementirali napreden algoritem, ki del podatkov hrani na vezju in z njimi učinkovito računa.

## 4.1 Definicija problema

Vzemimo matriko $A = [a_{i,j}]$ z dimenzijo $m \times n$, kjer je $m$ število vrstic in $n$ število stolpcev, ter matriko $B = [b_{i,j}]$ z dimenzijo $n \times l$. Rezultat množenja $A \times B$ je matrika $C = [c_{i,j}]$ z dimenzijo $m \times l$ in elementi

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}, 1 \leq i \leq m, 1 \leq j \leq l. \tag{4.1}$$

Enačba 4.1 nam pove, da so elementi matrike $C$ rezultati množenj elemen-

tov vrstice matrike A z elementi stolpca matrike B, nad katerimi se naredi vsota:

$$c_{i,j} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \cdots + a_{in} \cdot b_{nj}$$

Izračunajmo število operacij. Za izračun enega elementa matrike je potrebnih $n$ množenj. Število množenj za vse elemente matrike $C$ je

$$T^*_{mat}(m, n, l) = mnl = \Theta(mnl).$$

Za izračun enega elementa je treba zmnožke sešteti ($n-1$ seštevanj). Število operacij seštevanja za vse elemente matrike $C$ je

$$T^+_{mat}(m, n) = mnl - nl \sim mnl = \Theta(mnl).$$

Skupno število operacij za izračun matričnega množenja je

$$\begin{aligned} T_{mat}(m, n) &= T^*_{vec}(m, n) + T^+_{vec}(m, n) \\ &= mnl + mnl - mn \sim 2mnl \\ &= \Theta(mnl). \end{aligned}$$

Od tu naprej se bomo zaradi lažjega eksperimentalnega ovrednotenja ukvarjali s kvadratnimi matrikami velikosti $n \times n$, s časovno asimptotično zahtevnostjo množenja $\Theta(n^3)$. Obstajajo algoritmi, ki imajo manjšo časovno asimptotično zahtevnost (npr. Strassenov algoritem), vendar presegajo obseg magistrske naloge. Poleg tega algoritmi pogosto zahtevajo, da imajo elementi matrik nasprotno vrednost [1], zaradi česar jih ni mogoče uporabiti za vse polkolobarje, s katerimi se bomo ukvarjali v poglavju 5. V razdelku 3.1 so se med množenjem ponovno uporabljali samo elementi vektorja. Pri matričnem množenju imamo večkratno uporabo tako elementov matrike $A$ kot matrike $B$. Tudi tukaj je treba delati vsoto, ki bo v podatkovno-pretokovnem delu potrebovala zanko.

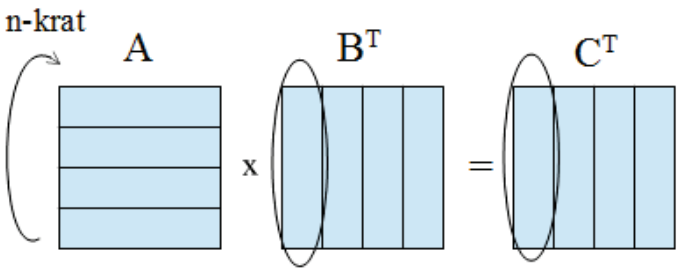
## 4.2 Ukazno-pretokovna implementacija

Razvili smo ukazno-pretokovno verzijo matričnega množenja za kvadratne matrike. Uporabljali jo bomo za testiranje pravilnosti rešitev za računalnik Maxeler in za primerjavo časov s kasnejšimi pospešenimi rešitvami. Algoritem je naslednji:

```
void multiply_CPU_matrix(int n, float *matA, float *matB, float *res){
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         float sum = 0;
         for (int k = 0; k < n; k++) {
            sum += matA[i*n+k] * matB[k*n+j];
         }
      res[i*n+j] = sum;
      }
   }
}
```

Algoritem je klasičen in množi elemente ene vrstice matrike $A$ z elementi enega stolpca matrike $B$ in nad zmnožki naredi seštevek za izračun enega elementa matrike $C$.

## 4.3 Osnovna rešitev




Slika 4.1: Matrično množenje in potrebni transponiranji

Osnovna rešitev matričnega množenja bo nadgradnja množenja in vektorja iz razdelka 3.4. Za matrično množenje algoritem nadgradimo tako, da

matriko $B$ razdelimo na vektorje in $n$-krat kličemo ščepec za množenje matrike in vektorja. Uporabimo lahko kar ščepec, implementiran v razdelku 3.4. Matriko $B$ je v ščepec potrebno poslati po stolpcih. To dosežemo tako, da jo transponiramo in pošljemo po vrsticah. Posledično tudi matriko $C$ na izhodu dobimo v transponirani obliki. Za pridobitev $C$ je zato, treba izhod še transponirati. Transponiranji zaradi branja in pisanja podatkov po stolpcih prikazuje slika 4.1.

Transponiranje in večkratni klic ščepca v ukazno-pretokovnem delu izvedemo na naslednji način:

```
transpose(n, matB, matBTrans);
for (int i=0; i<n; ++i) {
   vector = &matBTrans[n*i];
   MatMatMultiply(n, matA, vector, &output[n*i]);
}
transpose(n, output, outputTrans);
```

## 4.4 Opustitev transponiranja

V tej implementaciji lahko uporabimo ščepec, ki smo ga implementirali za algoritem množenja matrike in vektorja s transponiranjem. Enako kot v razdelku 3.5 je za izvajanje ščepca algoritma s transponiranjem treba transponirati levo matriko $A$. Tako kot v razdelku 4.3 je zaradi pošiljanja in prejemanja podatkov po stolpcih treba transponirati matriki $B$ in $C$. Ker velja

$$(A \cdot B)^\intercal = B^\intercal \cdot A^\intercal$$

in

$$(A^\intercal)^\intercal = A,$$

lahko v ščepec pošiljamo podatke za problem množenja $B^\intercal \cdot A^\intercal$ in s tem vsa 3 transponiranja izpustimo. Ščepec nam vrne

$$\begin{aligned}(B^\intercal \cdot A^\intercal)^\intercal &= ((A \cdot B)^\intercal)^\intercal \\ &= A \cdot B \\ &= C.\end{aligned}$$

V ukazno-pretokovnem delu torej transponiranji spustimo in zamenjamo vhoda matrike $A$ in matrike $B$.

```
for (int i=0; i<n; i++) {
   vector = &mat_a[n*i];
   MatMatMultiply(n*n, n, mat_b, vector, &output[n*i]);
}
```

## 4.5 Zmanjšanje odmika in uporaba cevi

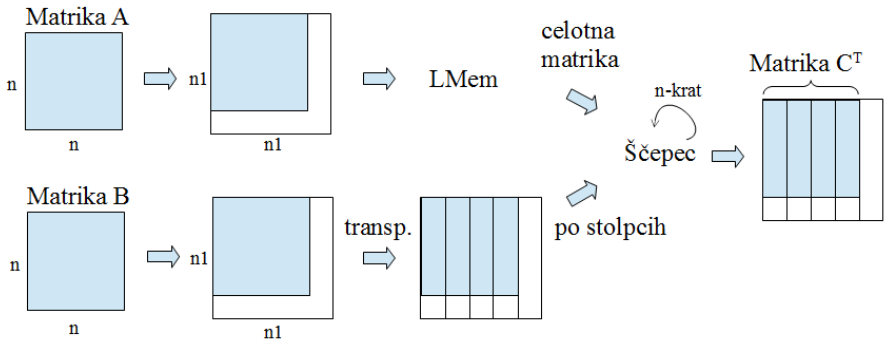
Algoritem matričnega množenja z zmanjšanjem odmika smo iz množenja matrike in vektorja v matrično množenje spremenili na enak način kot pri osnovnem algoritmu. Ščepec ostane enak kot v razdelku 3.6.

Enako je pri implementaciji cevi. Tako ščepci kot sprememba matrike ostanejo enaki, kot v razdelku 3.7. Vse tri algoritme kličemo na isti način, kot v razdelku 4.3.

## 4.6 Uporaba velikega pomnilnika LMem

Implementirani algoritmi matričnega množenja uporabljalo tako elemente matrike $A$ kot matrike $B$ večkrat (vsak element natanko $n$-krat). Vektor $b$ že zdaj shranimo v FMem in ga kličemo večkrat. Smiselno bi bilo pohitriti tudi branje matrike $A$. Za celotno matriko v FMemu ni dovolj prostora, na voljo pa imamo veliki pomnilnik LMem. Namesto da matriko vsakič pošiljamo iz ukazno-pretokovnega dela, jo shranimo v LMem, ki bo bližje našim ščepcem.

Slika 4.2 prikazuje potrebne prilagoditve za primer osnovnega algoritma. Podobno kot v razdelku 3.8 matriko $A$ z dodajanjem ničel spremenimo v velikost $n1 \times n1$, kjer je $n1$ deljiva s 96. Da bodo ščepci pravilno delovali, je enako treba prilagoditi matriko $B$. Izhod ščepca bo povečana matrika $C^T$, ki ji je treba odstraniti ničle in jo transponirati. Pri osnovnem algoritmu je dovolj, da ščepec kličemo $n$-krat, saj bodo zadnji stolpci ostali enaki nič. Pri ostalih algoritmih in pri uporabi cevi kličemo ščepec $n1$-krat.




Slika 4.2: Priprava podatkov za osnovni algoritem z LMemom

Izvorna koda za dodajanje ničel:

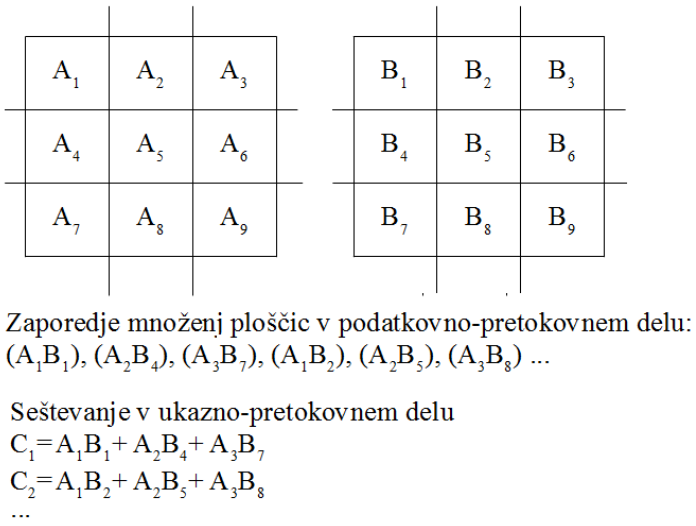
```
void align_matrix(int n, int n_aligned, float *mat, float *mat_aligned)
{
   for (int i = 0; i < n_aligned; i++) {
      for (int j = 0; j < n_aligned; j++) {
         if (i <n && j<n) {
            mat_aligned[i * n_aligned + j] = mat[i * n + j];
         }
         else {
            mat_aligned[i * n_aligned + j] = 0;
         }
      }
   }
}
```

in izvorna koda za odstranitev ničel.

```
void reverseAlignMatrix(int n, int n_aligned, float *mat_aligned, float
```

```
    *mat) {
      for (int i = 0; i < n_aligned; i++) {
      for (int j = 0; j < n_aligned; j++) {
         if (i <n && j<n) {
            mat[i * n + j] = mat_aligned[i * n_aligned + j];
         }
      }
   }
}
```

## 4.7 Razdelitev na ploščice




Slika 4.3: Razdelitev na ploščice

Še zadnja in najbolj napredna rešitev. Če želimo doseči maksimalne pohitritve, bi bilo idealno, da bi bila ena matrika vedno na čipu in bi se elementi druge ob prihodu lahko množili z vsemi potrebnimi elementi. Dejstvo je, da hitri pomnilnik vezja FMem ni dovolj velik, da bi lahko v celoti hranil veliko matriko. Do zdaj smo ga uporabili za hranjenje enega stolpca matrike $B$. Pretok med ukazno-pretokovnim in podatkovno-pretokovnim delom je bil še

vedno velik, saj je bilo celotno matriko $A$ treba v celoti brati $n$-krat. Prestavitev v LMem je to nekoliko omilila a tudi LMem dosega bistveno nižje hitrosti kot podatki na vezju. Za dodatno pohitritev bi bilo treba pretok do vezja čim bolj zmanjšati.

Nova rešitev je naslednja. Na vezju ni dovolj prostora za hranjenje celotne matrike, zato obe matriki razdelimo na manjše dele oziroma ploščice. Ploščico matrike $B$ bomo spravili na vezje in vsak element matrike $A$ množili s celotno ploščico matrike $B$. V ukazno-pretokovnem delu bomo ploščice ustrezno sešteli in sestavili v matriko $C$. Razdelitev in zaporedje množenj prikazuje slika 4.3. Med pisanjem magistrske naloge je bila podobna rešitev predlagana za redke matrike v [29]. Zgledovali smo se tudi po algoritmih iz [15].

Poglejmo si izvorno kodo ščepca:

```
1  MultiplyKernel(KernelParameters parameters, int tileSize) {
2    super(parameters);
3
4    CounterChain cc = control.count.makeCounterChain();
5    DFEVar i = cc.addCounter(tileSize, 1);
6    DFEVar j = cc.addCounter(tileSize, 1);
7
8    DFEVar[] summands = new DFEVar[tileSize];
9
10   DFEVar aIn = io.input("matA", dfeFloat(8,24));
11   DFEVar bIn = io.input("matB", dfeFloat(8,24));
12
13   for (int n = 0; n < tileSize; ++n) {
14     DFEVar a = Reductions.streamHold(stream.offset(aIn, n), (j === 0));
15     DFEType addrType = dfeUInt(MathUtils.bitsToAddress(tileSize));
16     DFEVar address = j.cast(addrType);
17     Memory<DFEVar> bBuf = mem.alloc(bIn.getType(), tileSize);
18     bBuf.write(address, bIn, (i === n));
19     DFEVar b = stream.offset(bBuf.read(address), tileSize*tileSize);
20     summands[n] = a * b;
21   }
22   io.output("output", TreeReduce.reduce(new Add<DFEVar>(), summands),
```

```
        scalarType);
23  }
```

V vrsticah od 4 do 6 inicializiramo verigo dveh števcev, ki bosta štela od 0 do velikosti ploščice minus ena. V 8. vrstici inicializiramo tabelo, ki bo hranila vmesne zmnožke. Nato v vrsticah 10 in 11 nastavimo vhoda, na katera bodo prišli elementi ploščice matrike $A$ in ploščice matrike $B$.

V 13. vrstici bomo z zanko $for$ paralelizirali naš program. Ne gre za tipično zanko, ampak za način, s katerim bomo na nivoju strojne opreme pomnožili vsa vozlišča znotraj zanke $for$, ta pa se bodo izvajala sočasno. Znotraj zanke $for$ bomo poskušali doseči, da se v eni urini periodi izračunajo zmnožki ene vrstice ploščice $A$ in enega stolpca ploščice $B$. V prvi urini periodi bomo množili 1. vrstico ploščice $A$ s prvim stolpcem ploščice $B$. V naslednji urini periodi bomo 1. vrstico ploščice $A$ množili z drugim stolpcem ploščice $B$. Po $tileSize$ urinih periodah se bomo pomaknili v novo vrstico $A$ in začeli ponovno s 1. stolpcem $B$.

V 14. vrstici bomo tako pomnoženim spremenljivkam $a$ (torej $a1$, $a2$ ...) določili po en element vrstice $A$ in ga nato naslednjih $tileSize$ period ne bomo spreminjali. To dosežemo z $Reductions.streamHold$, ki vrača ali trenutno ali prejšnjo vrednost toka. Vozlišče sprejeme kot prvi parameter tok in kot drugi parameter stikalo $store$. Shranjevanje deluje na naslednji način:

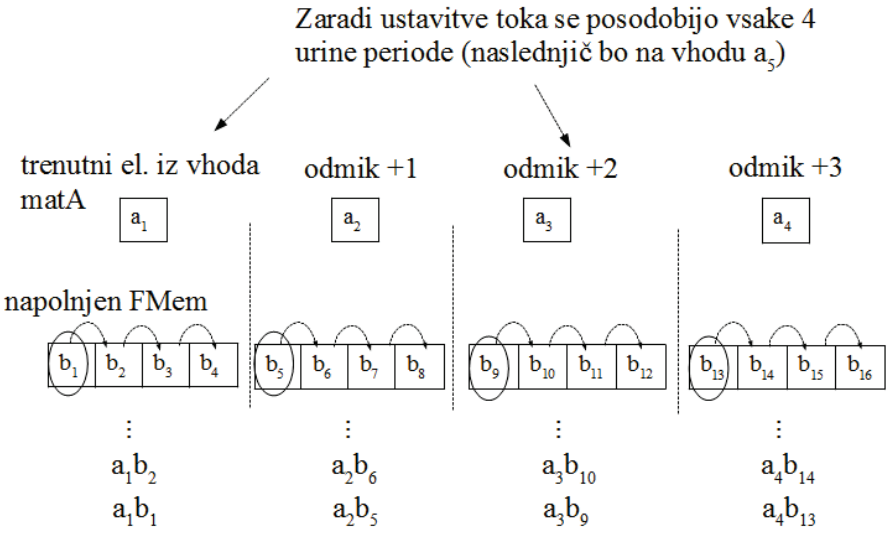
- V primeru, da je $store$ 1, vozlišče shrani trenutno vrednost toka.
- V primeru, da je $store$ 0, ignorira trenutno vrednost toka.

Izhodne vrednosti so naslednje:

- V primeru, da je $store$ 1, vozlišče vrne trenutno vrednost toka.
- V primeru, da je $store$ 0, vozlišče vrne trenutno shranjeno vrednost[16].

$Reductions.streamHold$ bo torej za vsak $j = 0$ shranjeval vrednost. V kombinaciji z vozliščem $offset$ bo v 1. urini periodi v vsako izmed vzporednih vozlišč ($a1$, $a2$ ...) shranil po en element vrstice. Naslednjih $tileSize - 1$ period bodo vozlišča ostala enaka, nato pa bodo dobila elemente nove vrstice.

V vrsticah od 17 do 19 shranimo celotno ploščico $B$ na čip. Najprej v 17. vrstici v vsakem izmed vzporednih stez dodelimo pomnilnik velikosti ene vrstice ploščice. Pisali bomo enkrat na urino periodo, in sicer v prvih $tileSize$ periodah zapišemo v 1. vzporedno enoto FMema prvo vrstico ploščice $B$. V naslednjih $tileSize$ periodah zapišemo v 2. enoto FMema 2. vrstico ploščice $B$ in tako naprej do zadnje. Končno stanje FMema in prehod elementov ploščice $A$ prikazuje slika 4.4.




Slika 4.4: Prehod elementov za velikost ploščice 4

V 19. vrstici se bere vsako urino periodo. V FMemu bodo vsi elementi na voljo šele po $tileSize * tileSize$ urinih periodah, zato uporabimo $stream.offset$ in beremo iz tam. Sočasno beremo po en element iz vsakega pomnilnika in v eni urini periodi dobimo celoten stolpec ploščice $B$. V 20. vrstici elemente zmnožimo. Nato v 22. vrstici elemente s funkcijo $TreeReduce.reduce$ seštejemo in pošljemo na izhod.

## 4.8 Eksperimentalno ovrednotenje

### 4.8.1 Postavitev eksperimenta

Programska in strojna oprema je enaka kot v razdelku 3.9.

| **Ime algoritma** | **Povzetek delovanja** |
|---|---|
| cpu | Ukazno-pretokovni algoritem za matrično množenje, implementiran v programskem jeziku C. |
| Basic | Osnovno matrično množenje. Podatkovno-pretokovni algoritem, kjer je koreografija pretakanja podatkov podobna ukazno-pretokovni implementaciji in je vmes treba čakati na delno vsoto. |
| Transposed | Algoritem z opustitvijo transponiranja. Uporablja ščepec algoritma množenja matrike in vektorja s transponiranjem. V tem primeru se transponiranj znebimo z zamenjavo vhodnih matrik. |
| Tiled | Algoritem z zmanjšanjem odmika, ki uporablja ščepec množenja matrike in vektorja z zmanjšanjem odmika. |
| <Algoritem>Vec<št.><br><LMem> | Implementacija s cevmi za izbrani algoritem in izbrano število cevi. Značka LMem nakazuje uporabo LMema. |
| PagingAlg<št.> | Algoritem z razdelitvijo na ploščice, kjer matriki po delih spravimo v podatkovno-pretokovni del in z njimi učinkovito računamo. Število nam pove velikost ploščice. |

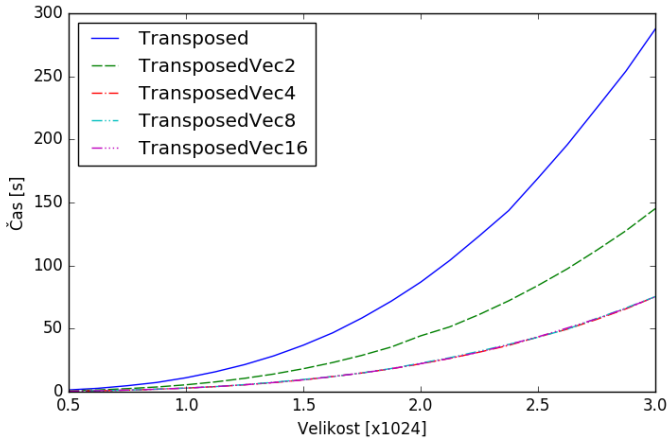
Tabela 4.1: Pregled testiranih algoritmov matričnega množenja skupaj s povzetki delovanja

Testirane algoritme in povzetke delovanj prikazuje tabela 4.1. Testna

množica za vse algoritme, razen algoritma z razdelitvijo na ploščice, so bile matrike velikosti od $512 \times 512$ do $3072 \times 3072$ v korakih po 128. Matriki sta generirani naključno z elementi velikosti od $-200$ do 200. Osnovno matrično množenje smo zaradi počasnosti pri testih izpustili in se osredotočili na ostale algoritme. Testirali smo do maksimalno 96 cevi. Pri algoritmu z razdelitvijo na ploščice smo testirali na matrikah velikosti od $512 \times 512$ do $10240 \times 10240$ v korakih po 512.

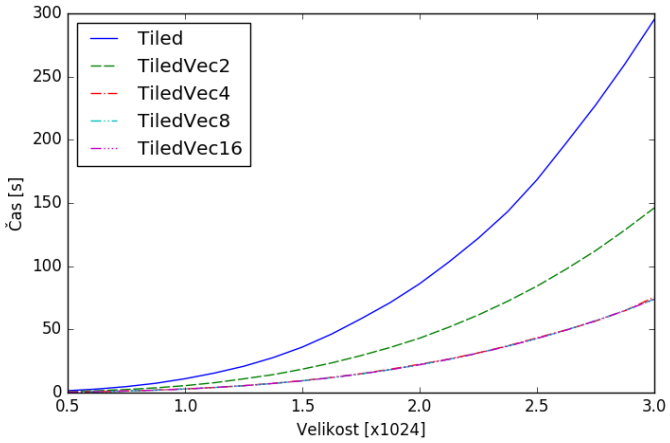
### 4.8.2 Rezultati

Sliki 4.5 in 4.6 prikazujeta čase za algoritme matričnega množenje s transponiranjem in z zmanjšanjem odmika. Podobno kot pri množenju matrike in vektorja opazimo, da izboljšave dosegamo vse do štirih cevi. Za večje število cevi zaradi premajhnega pretoka podatkov med ukazno-pretokovnim in podatkovno-pretokovnim delom časi ostanejo enaki. Nadgradnjo teh




Slika 4.5: Primerjava časa izvajanja za algoritme matričnega množenja s transponiranjem pri različnem številu cevi

dveh algoritmom z uporabo LMema prikazujeta sliki 4.7 in 4.8. Uporaba LMema je bistveno izboljšala rezultate. Dosegli smo približno 50-kratne pohitritve v primerjavi z algoritmom brez cevi in približno 4-kratne pohitritve v primerjavi z algoritmi s cevmi in brez LMema.

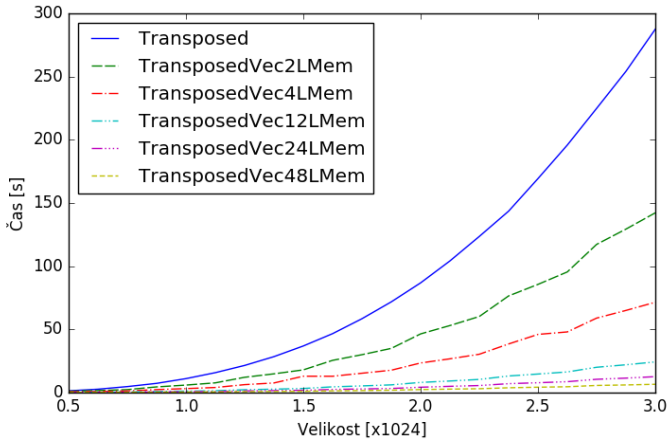



Slika 4.6: Primerjava časa izvajanja za algoritme matričnega množenja z zmanjšanjem odmika pri različnem številu cevi

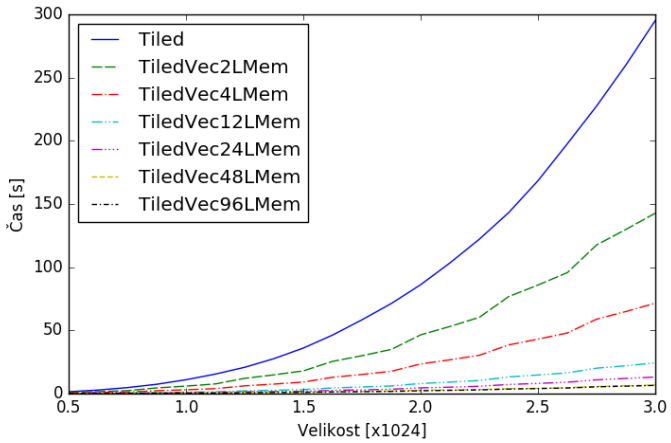
Pri množenju matrike in vektorja je bil algoritem z zmanjšanjem odmika hitrejši, tu pa bistvenih razlik v hitrosti ni. Razlog je morda v tem, da tu delamo z manjšimi matrikami (3072 namesto 16384) in sam odmik ni več tako kritičen. Algoritem z zmanjšanjem odmika smo testirali tudi s 96 cevmi in v primerjavi z algoritmom z 48 cevmi ni bilo pohitritev, kar nakazuje na omejitve prepustnosti LMema. Če bi bila ta večja, bi bil algoritem še hitrejši.

Slika 4.9 prikazuje primerjavo izbranih hitrejših podatkovno-pretokovnih algoritmov in ukazno-pretokovnega algoritma. Algoritma z 48 cevmi sta približno 5-krat hitrejša od ukazno-pretokovne implementacije. Za slednjega na grafu ni prikazanih časov pri velikostih 2048, 2560 in 3072. Uporabljen procesor pri teh vrednostih naleti na veliko konfliktov v predpomnilniku in se zaradi tega odreže občutno slabše (nekajkrat počasnejše). Gre za izjemne primere, zato smo jih za boljšo predstavitev tipičnih časov umaknili.

Algoritem z razdelitvijo na ploščice je dosegel bistveno boljše pohitritve in ga prikazuje slika 4.10. Uspelo nam ga je prevesti vse do velikosti ploščice 384. Večja kot je ploščica, hitreje se program izvede. Občutne izboljšave se dosegajo do ploščice velikosti 256, potem pa so pohitritve vedno bolj zane-



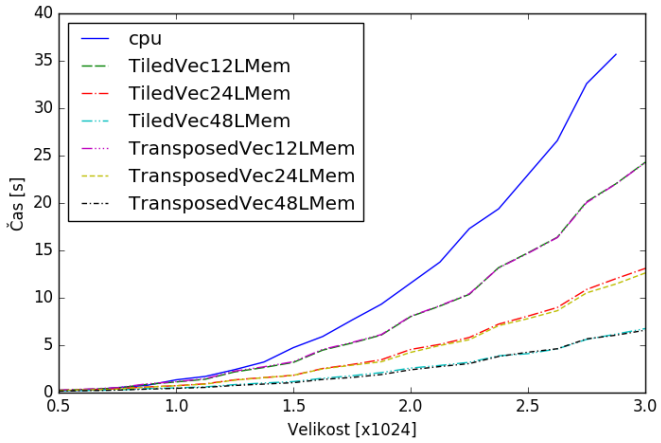

Slika 4.7: Primerjava časa izvajanja za algoritme matričnega množenja s transponiranjem in z uporabo LMema pri različnem številu cevi



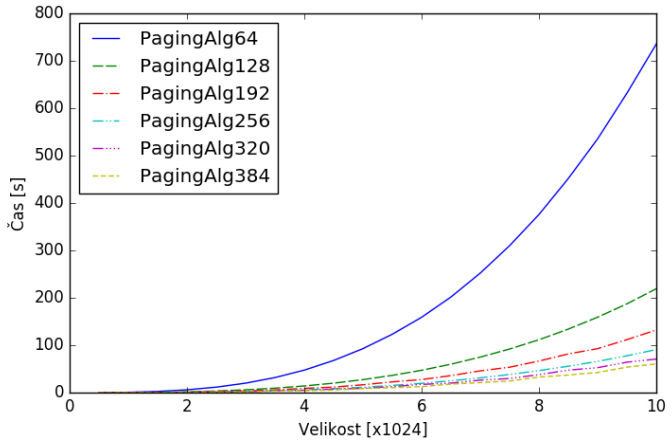

Slika 4.8: Primerjava časa izvajanja za algoritme matričnega množenja z zmanjšanjem odmika in z uporabo LMema pri različnem številu cevi

marljive. Če bi uporabljali sredstva FPGA za druge operacije (pri sorodnem problemu), bi lahko bilo smiselno vzeti manjšo velikost ploščice od maksimalne.

Faktor pohitritve algoritma z zmanjšanjem odmika in 48 cevmi ter algoritma s ploščicami v primerjavi z ukazno-pretokovno različico prikazuje slika 4.11. Pohitritve algoritma z zmanjšanjem odmika so približno 30-kratne,

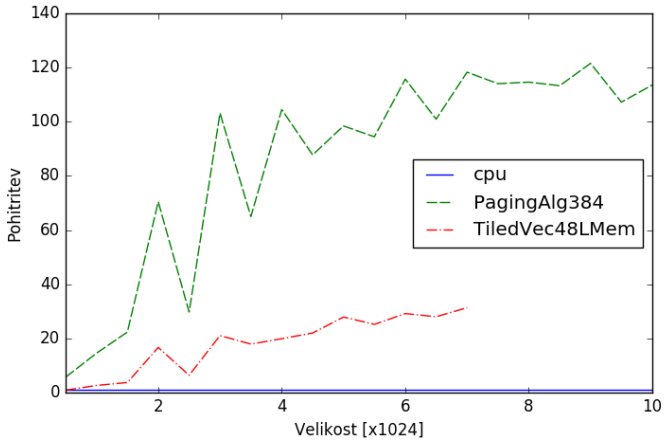



Slika 4.9: Primerjava hitrejših algoritmov skupaj z ukazno-pretokovnim algoritmom




Slika 4.10: Primerjava časa izvajanja za algoritme matričnega množenja z razdelitvijo na ploščice pri različno velikih ploščicah

medtem ko so pohitritve algoritma z razdelitvijo na ploščice tudi do 110-kratne. V začetku grafa opazimo zelo različne pohitritve. V tem grafu nismo umaknili vrednosti, pri katerih se procesor zaradi veliko zgrešitev pri dostopu do predpomnilnika odreže občutno slabše.




Slika 4.11: Faktor pohitritve algoritma z zmanjšanjem odmika, algoritma z razdelitvijo na ploščice in ukazno-pretokovnega algoritma

### 4.8.3 Poraba sredstev vezja FPGA

Porabo sredstev vezja FPGA v procentih prikazujejo tabele 4.2, 4.3 in 4.4. Pregledno primerjavo prikazujejo histogrami 4.12, 4.13 in 4.14. Še kratka analiza. Algoritem s transponiranjem in algoritem z zmanjšanjem odmika pri 48 ceveh dosežeta maksimalne hitrosti in se z dodatnimi cevmi ne pohitrita več. Pri 48 ceveh na vezju ostane še nekaj prostora za morebitne dodatne operacije (pri sorodnem problemu). Algoritem s ploščicami zaradi velike paralelnosti v največji obliki vzame kar 40 % signalnih procesorjev DSP (angl. digital signal processor), s katerimi vzporedno računa. S tem je za matrično množenje zelo optimiziran za morebitne dodatne operacije pa je nekoliko manj prostora.

| Algoritem | LUT | FF1 | FF2 | BRAM | DSP |
|---|---|---|---|---|---|
| Transposed | 2,33 | 2,16 | 0,58 | 1,69 | 0,10 |
| TransposedVec2 | 2,69 | 2,55 | 0,67 | 2,16 | 0,20 |
| TransposedVec4 | 3,22 | 3,41 | 0,64 | 2,77 | 0,40 |
| TransposedVec8 | 4,75 | 4,97 | 1,05 | 5,03 | 0,79 |
| TransposedVec16 | 7,50 | 8,42 | 1,44 | 8,69 | 1,59 |
| TransposedVec2LMem | 11,55 | 14,22 | 2,94 | 11,00 | 0,20 |
| TransposedVec4LMem | 11,99 | 15,20 | 2,80 | 11,80 | 0,40 |
| TransposedVec12LMem | 15,37 | 18,42 | 3,59 | 15,65 | 1,19 |
| TransposedVec24LMem | 19,63 | 23,11 | 4,66 | 21,19 | 2,38 |
| TransposedVec48LMem | 28,10 | 32,11 | 6,63 | 30,17 | 4,76 |
| TransposedVec96LMem | 44,35 | 52,61 | 9,27 | 54,28 | 9,52 |

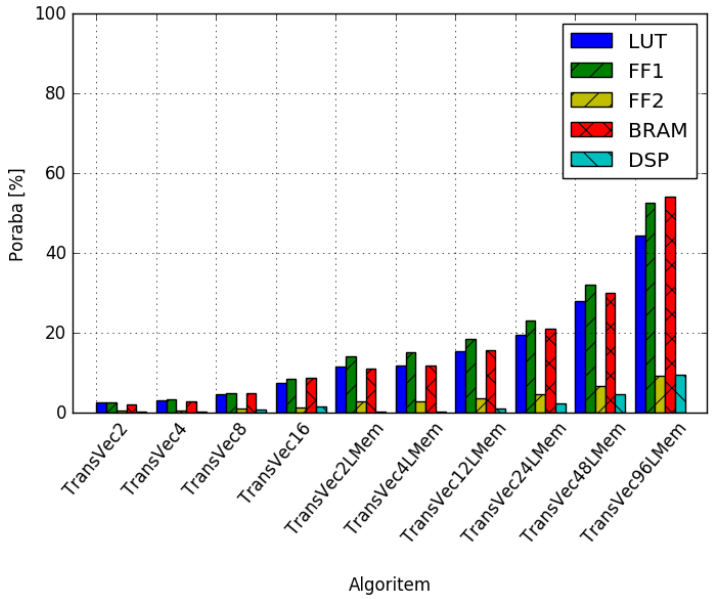
Tabela 4.2: Poraba sredstev vezja FPGA [%] za algoritme s transponiranjem.

| Algoritem | LUT | FF1 | FF2 | BRAM | DSP |
|---|---|---|---|---|---|
| Tiled | 2,32 | 2,12 | 0,59 | 1,74 | 0,10 |
| TiledVec2 | 2,59 | 2,66 | 0,54 | 2,30 | 0,20 |
| TiledVec4 | 3,34 | 3,17 | 0,91 | 2,82 | 0,40 |
| TiledVec8 | 4,90 | 5,07 | 1,07 | 5,08 | 0,79 |
| TiledVec16 | 7,94 | 8,50 | 1,68 | 8,74 | 1,59 |
| TiledVec2LMem | 11,39 | 14,39 | 2,74 | 11,14 | 0,20 |
| TiledVec4LMem | 12,25 | 14,97 | 3,06 | 11,84 | 0,40 |
| TiledVec12LMem | 15,38 | 18,40 | 3,63 | 15,74 | 1,19 |
| TiledVec24LMem | 19,76 | 23,45 | 4,62 | 21,29 | 2,38 |
| TiledVec48LMem | 28,52 | 33,54 | 6,11 | 30,26 | 4,76 |
| TiledVec96LMem | 46,35 | 54,27 | 9,70 | 54,37 | 9,52 |

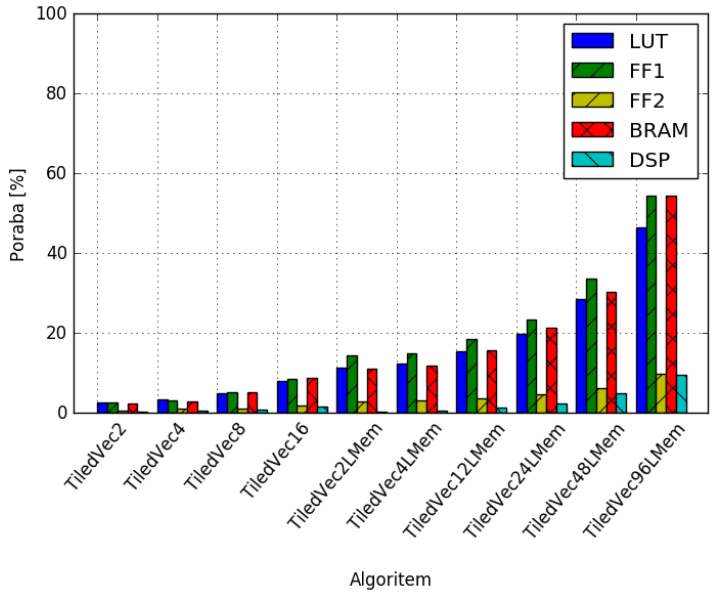
Tabela 4.3: Poraba sredstev vezja FPGA [%] za algoritme z zmanjšanjem odmika.

| Algoritem | LUT | FF1 | FF2 | BRAM | DSP |
|---|---|---|---|---|---|
| PagingAlg64 | 13,84 | 16,48 | 2,23 | 4,42 | 6,35 |
| PagingAlg128 | 25,97 | 30,78 | 4,43 | 7,66 | 12,70 |
| PagingAlg192 | 38,50 | 44,98 | 6,69 | 11,18 | 19,05 |
| PagingAlg256 | 50,64 | 58,98 | 9,36 | 14,85 | 25,40 |
| PagingAlg320 | 63,78 | 70,02 | 14,64 | 18,70 | 31,75 |
| PagingAlg384 | 77,67 | 80,48 | 20,89 | 22,74 | 38,10 |

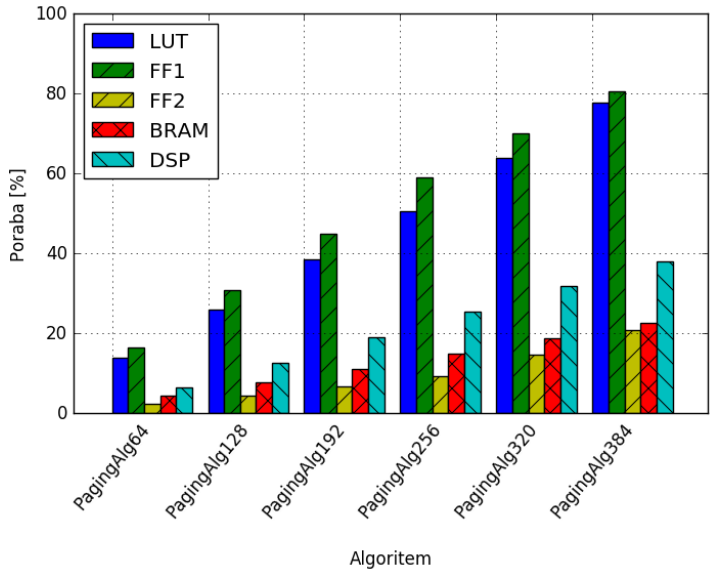
Tabela 4.4: Poraba sredstev vezja FPGA [%] za algoritme z razdelitvjo na ploščice.




Slika 4.12: Grafični prikaz primerjave porabe sredstev vezja FPGA za algoritme s transponiranjem




Slika 4.13: Grafični prikaz primerjave porabe sredstev vezja FPGA za algoritme z zmanjšanjem odmika




Slika 4.14: Grafični prikaz primerjave porabe sredstev vezja FPGA za algoritme z razdelitvijo na ploščice

# Poglavje 5

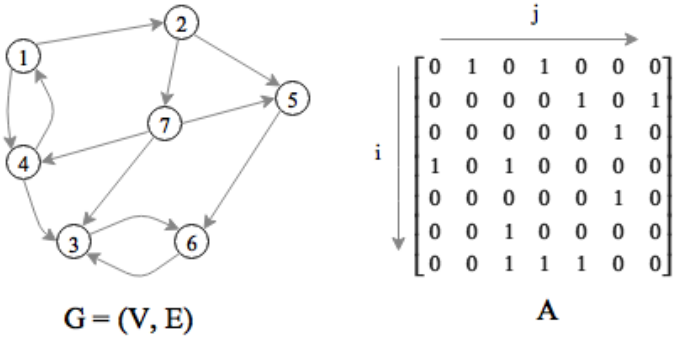
# Razširitev na ostale matrične algoritme

Ideja tega poglavja je, da algoritme, implementirane in pohitrene v poglavjih 3 in 4, uporabimo za reševanje novih računalniških problemov. Glede na to, da naši algoritmi dosegajo zelo dobre rezultate, jih je seveda smiselno uporabiti tudi drugje. Predvsem nas bodo zanimali algoritmi na grafih in računanje v polkolobarju. Implementirali in eksperimentalno ovrednotili bomo algoritem za potenciranje matrik in algoritem iskanja najkrajših poti med vsemi pari vozlišč.

## 5.1 Matrika sosednosti

Osnova za pretvorbo matričnih algoritmov v algoritme na grafih je možnost alternativnega zapisa grafov; namesto s parom vozlišč in povezav lahko graf zapišemo z matriko. Osnovni koncept, ki je prikazan na sliki 5.1, je matrika sosednosti in je del teorije grafov že od začetka računalništva [8]. Usmerjen graf $G = (V, E)$ z vozlišči $V = \{1, 2, ..., n\}$ lahko zapišemo z matriko $A$ reda $n \times n$ in elementi $A(i, j) = 1$, če obstaja povezava med $i$ in $j$, sicer $A(i, j) = 0$.

Zanimiva je tudi povezava med množenjem matrike in vektorja ter algo-




Slika 5.1: Graf, prikazan z množico vozlišč in povezav, ter pripadajoča matrika sosednosti

ritmom iskanja v širino. Množenje

$$A^T v,$$

kjer je $A$ matrika sosednosti in $v$ vektor z ničlami povsod, razen na $s$-tem mestu, nam vrne vse sosede vozlišča $s$, do katerih je mogoče priti v enem koraku [2]. Vse algoritme iz poglavja 3 lahko torej brez prilagoditev uporabimo za iskanje sosedov vozlišča.

## 5.2 Polkolobarji

Nadalje lahko matrične algoritme spremenimo v algoritme za reševanje problemov grafov tako, da jim zamenjamo operaciji seštevanja in množenja z novima operacijama. Da bo množenje še vedno delovalo pravilno, to ne moreta biti poljubni operaciji. Izbrani operaciji imata manj stroge pogoje kot na primer klasično množenje in seštevanje v $\mathbb{R}$ (ki je kolobar). Izkaže se, da veliko problemov rešujejo operacije, ki v kombinaciji z množico, na kateri se izvajajo, ustrezajo ravno lastnostnim polkolobarja. Polkolobar je neprazna množica $R$ z dvema binarnima operacijama, ki ju imenujemo seštevanje (označeno s $\oplus$) in množenje (označeno s $\odot$) in velja [5]:

- Operacija seštevanja je asociativna, komutativna in ima nevtralni element $\overline{0}$.
- Operacija množenja je asociativna in ima nevtralni element $\overline{1}$.
- Operaciji množenja in seštevanja povezujeta leva in desna distributivnost.
- Nevtralni element množenja $\overline{0}$ je absorpcijski. To pomeni da $r \odot \overline{0} = \overline{0} = \overline{0} \odot r$ za vse $r \in R$ .

Operacijama $\oplus$ in $\odot$ lahko torej določimo poljubni novi operaciji, ki ustrezata zgornjim pogojem, in s tem rešimo veliko novih problemov. Pri tem dobita tudi $\overline{1}$ in $\overline{0}$ zamenjavo, ki ustreza zgornji definiciji. Za nas pride v poštev uporaba polkolobarjev v kombinaciji z matričnim množenjem (ki smo ga najbolj pohitrili). Nekaj takih primerov in njihovo uporabo prikazuje tabela 5.1, ki je povzeta po [28]. Primer uporabe tabele prikazuje algoritem za iskanje najkrajših poti med vsemi pari vozlišč, ki je implementiran v 5.5.1.

| $R$ | $\oplus$ | $\odot$ | $\overline{0}$ | $\overline{1}$ | Uporaba |
|---|---|---|---|---|---|
| $\mathbb{R}$ | $+$ | $\cdot$ | 0 | 1 | Množenje realnih matrik |
| $(0,1)$ | $\vee$ | $\wedge$ | 0 | 1 | Dosegljivost vozlišč |
| $\mathbb{R} \cup \infty$ | $min$ | $+$ | $\infty$ | 0 | Najkrajše poti med vsemi pari vozlišč |
| $\mathbb{R} \cup \{-\infty\}$ | $max$ | $+$ | $-\infty$ | 0 | Iskanje kritične poti |
| $\mathbb{R}^+ \cup \{\infty\}$ | $min$ | $max$ | $\infty$ | 0 | Minimalno vpeto drevo |
| $\mathbb{R}^+ \cup \{\infty\}$ | $max$ | $min$ | 0 | $\infty$ | Problem najširše poti |

Tabela 5.1: Polkolobarji matričnih množenj in njihova uporaba

## 5.3 Potenciranje matrik

Pred tem si poglejmo, kako lahko matrično množenje nadgradimo v algoritem za potenciranje matrik. Najprej pozabimo arhitekturo in si poglejmo dve

možnosti za potenciranje matrik. Po definiciji gre pri potenciranju za navadno ponavljanje množenj z elementom ki ga potenciramo. Potenco $B = A^n$ lahko tako izračunamo z naslednjo zanko $for$:

$B \leftarrow A$
**for** i = 2 : n **do**
    $B = A \cdot B$
**end for**

V primeru, da ne potrebujemo vmesnih rezultatov, lahko $A^n$ izračunamo tudi hitreje. Algoritem izpeljemo iz znanega algoritma za potenciranje s pomočjo kvadriranja. Namesto da vsakič množimo z osnovno matriko, lahko uporabimo že potencirano matriko. Iz

$$x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{če je } n \text{ lih} \\ (x^2)^{\frac{n}{2}}, & \text{če je } n \text{ sod} \end{cases}$$

lahko izpeljemo algoritem, ki bo matriko vsakič ali kvadriral ali pa kvadriral in množil.

**Algoritem 1:** Potenciranje matrik s kvadriranjem

```
1:  first ← True
2:  while n > 1 do
3:      if n sod then
4:          A = A · A
5:          n = n/2
6:      else
7:          if first then
8:              B ← A
9:              first ← False
10:         else
11:             B ← A · B
12:         end if
13:         A = A · A
```

14: $n = (n-1)/2$
15: **end if**
16: **end while**
17: **if** first **then**
18: $rezultat \leftarrow A$
19: **else**
20: $rezultat \leftarrow A \cdot B$
21: **end if**

| Korak | n | Operacije | A | B |
|---|---|---|---|---|
| 1. | $n = 13$ | $B = A, A = A \cdot A$ | $A^2$ | $A$ |
| 2. | $n = 6$ | $A = A \cdot A$ | $A^4$ | $A$ |
| 3. | $n = 3$ | $B = A \cdot B, A = A \cdot A$ | $A^8$ | $A^5$ |
| 4. | $n = 1$ | $rezultat = A \cdot B = A^{13}$ | | |

Tabela 5.2: Postopek izvajanja algoritma potenciranja s kvadriranjem za potenco velikosti 13

V tabeli 5.2 vidimo postopek izvajanja algoritma za potenco reda 13. Algoritem bo namesto dvanajstih izvedel šest matričnih množenj. V splošnem bo namesto $n-1$ množenj izvedel $\lfloor \log_2 n \rfloor$ kvadriranj $A$ in največ $\lfloor \log_2 n \rfloor$ množenj $A \cdot B$, skupaj torej največ $2 \lfloor \log_2 n \rfloor$ matričnih množenj.

### 5.3.1 Podatkovno-pretokovna implementacija

Nadgradnja matričnega množenja iz razdelka 4.7 v potenciranje matrik je relativno preprosta. Zagotovo pa je zanimiva iz praktičnega vidika. Uporablja se na primer pri algoritmu v naslednjem razdelku. Za implementacijo rešitve definiramo v ukazno-pretokovnem delu funkcijo

```
mat_multiply(float *mat_a, float *mat_b, float *mat_c),
```

ki bo s pomočjo podatkovno-pretokovnega računalnika zmnožila dve matriki na enak način kot v razdelku 4.7. Zgornja algoritma implementiramo

v ukazno-pretokovnem delu in za matrično množenje uporabimo funkcijo *mat_multiply*().

## 5.4 Iskanje sprehodov podane dolžine

Potenciranje matrike sosednosti reši nov problem. Potenca $A^n$ je ravno matrika, katere element $A(i, j)$ predstavlja število možnih sprehodov dolžine $n$ iz $i$ v $j$. Zakaj je tako? Poglejmo si matriko sosednosti $A$. Njeni elementi vsebujejo število sprehodov dolžine 1 (to so povezave grafa), saj imamo 1 le na mestih, kjer povezava obstaja. Indukcijsko predpostavimo, da elementi $A^k$ podajajo števila sprehodov dolžine $k$, in si poglejmo $A^{k+1}$. Velja $A^{k+1} = A^k A$. Za poljuben element $A(i, j)$ izračunamo vrednost $A^{k+1}(i, j)$ na naslednji način:

$$A^{k+1}(i, j) = \sum_{l=1}^{n} A^k(i, l)A(l, j).$$

Pri tem je $l$ vmesno vozlišče med $i$ in $j$. Vrednost $A^k(i, j)$ je po indukcijski predpostavki število sprehodov dolžine $k$ med $i$ in $l$. $A(l, j)$ je 1 za elemente, kjer obstaja povezava med $l$ in $j$. Vsota torej prešteje vse sprehode dolžine $k$ iz vozlišča $i$ v $l$, kjer ima $l$ neposredno povezavo z $j$. Sklenemo lahko, da $A^k(i, j)$ podaja število sprehodov dolžine $k + 1$ iz $i$ v $j$ [19].

### 5.4.1 Iskanje trikotnikov

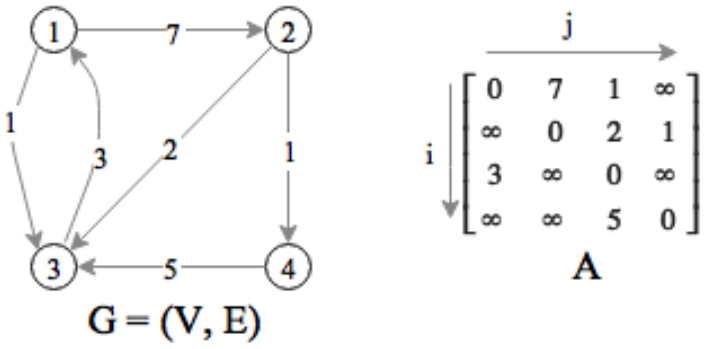
Naslednji algoritem, ki ga lahko izpeljemo, je iskanje števila trikotnikov v neusmerjenemu grafu. Trikotniki so cikli dolžine tri. Torej so to poti iz nekega vozlišča nazaj v isto vozlišče v natanko treh korakih. To lastnost bodo imeli ravno diagonalni elementi matrike $A^3$. Matrika $A^3$ vsebuje število vseh možnih sprehodov med dvema vozliščema. Nas zanima samo število trikotnikov, ne pa tudi sama pot ter začetek in konec vozlišča. Potenciranje $A^3$ vsak trikotnik, začet iz določenega vozlišča, šteje dvakrat, ker se da po ciklu potovati v obe smeri. Poleg tega se vsak trikotnik šteje trikrat, saj

je lahko začetek cikla vsaka njegova točka. Za izračun je torej treba sešteti diagonalo matrike $A^3$ in jo deliti s šest [19].

Tako algoritem iskanja prehodov podane dolžine in iskanja trikotnikov lahko s pomočjo algoritmov iz poglavja 4 trivialno prenesemo na podatkovno-pretokovno arhitekturo in občutno pohitrimo.

## 5.5 Iskanje najkrajših poti med vsemi pari vozlišč

Vrnimo se nazaj na polkolobarje in rešimo problem iskanja najkrajših poti med vsemi pari vozlišč ali problem APSP (angl. all-pairs shortest path). Problem je naslednji: imamo usmerjen graf $G$ z $n$ vozlišči in $m$ povezavami ter matriko cen $A$ z elementi $A(i, j)$, ki vsebujejo ceno sprehoda iz vozlišča $i$ v $j$. Zanima nas nova matrika $B$ z elementi $B(i, j)$, ki nam povejo najmanjšo ceno sprehoda iz $i$ v $j$, pri čemer število sprehodov ni pomembno. Primer grafa in pripadajoče matrike $A$ prikazuje slika 5.2. Elementi $A(i, i)$ matrike so enaki 0, saj je cena, da pridemo do vozlišča, v katerem smo že, enaka 0. Elementi, ki nimajo direktnih povezav, imajo ceno sprehoda $\infty$.




Slika 5.2: Graf in pripadajoča matrika cen

### 5.5.1 Implementacija z linearno algebro

Uporabimo teorijo iz razdelka 5.2. Natančneje izberemo polkolobar $(\mathbb{R}^+ \cup \infty, min, +)$ s katerim lahko rešimo problem iskanja najkrajših poti med vsemi pari vozlišč in algoritem izvedemo s spremenjenim matričnim množenjem. Operacijo seštevanja bomo nadomestili z minimumom elementov, operacijo množenja pa z običajnem seštevanju v $\mathbb{R}$. Nova matrika $C = A \odot B$ bo imela elemente

$$c(i,j) = \min_{k=1}^{n}\{a(i,k) + b(k,j)\}.$$

Poglejmo si, kaj se zgodi z množenjem $A \odot A$. Operacija $\min_{k=1}^{n}\{a_{i,k} + b_{k,j}\}$ bo za vse $k$ seštela ceno premika iz vozlišča $i$ v $k$ in iz $k$ v $j$ ter izbrala najcenejšo. Matrika $A^2$ bo torej vsebovala najcenejše premike iz $i$ v $j$ v največ dveh korakih. Nadaljnje množenje z $A$ bo izračunalo najcenejše premike med dvema vozliščema v treh korakih. V $A^{n-1}$ bomo dobili matriko z elementi, ki bodo predstavljali minimalne cene med vozlišči v $n-1$ korakih, in hkrati tudi končne minimalne cene sprehodov med vozlišči. Za tem se tudi z dodatnim množenjem matrika ne bo več spreminjala. Nadaljnji opis in dokazi so na voljo v [6]. Namesto $n-2$ zaporednih množenj matriko kvadriramo do najmanj $A^{n-1}$. Algoritem za rešitev problema je tako:

$i \leftarrow 1$
**while** i < n-1 **do**
  $A = A \odot A$
  $i \leftarrow i * 2$
**end while**

Algoritem ima asimptotično časovno zahtevnost $\Theta(n^3 \log n)$ [31].

### 5.5.2 Floyd-Warshall

Algoritem Floyd-Warshall rešuje problem APSP s časovno asimptotično zahtevnostjo $\Theta(n^3)$ [31]. Implementirali smo ga v ukazno-pretokovni različici. Algoritem (za matriko cen $A$), opisan v [4], je naslednji:

**Algoritem 2:** Floyd-Warshall

```
1: for k ← 1, n do
2:     for i ← 1, n do
3:         for j ← 1, n do
4:             if A[i][j] > A[i][k] + A[k][j] then
5:                 A[i][j] ← A[i][k] + A[k][j]
6:             end if
7:         end for
8:     end for
9: end for
```

Algoritem nas bo zanimal za primerjavo z algoritmom s slabšo asimptotično časovno zahtevnostjo, a implementiranim na podatkovno-pretokovni arhitekturi.

### 5.5.3 Podatkovno-pretokovna implementacija

Za prenos na podatkovno-pretokovno arhitekturo je primernejša implementacija z linearno algebro. Sicer ima slabšo asimptotično časovno zahtevnost, omogoča pa več paralelnosti [6]. Nadgradili smo različico matričnega množenja z razdelitvijo na ploščice, saj se je izkazala za najhitrejšo. Prilagoditev izvedemo na naslednji način. Najprej v ščepcu, kjer se izvajajo množenja posameznih ploščic matrike, spremenimo operacijo množenja v seštevanje. Vrstico

```
summands[n] = a * b;
```

spremenimo v

```
summands[n] = a + b;
```

Namesto vsote bomo delali minimum. Vrstico

```
io.output("output", TreeReduce.reduce(new Add<DFEVar>(), summands),
    scalarType);
```

spremenimo v

```
io.output("output", TreeReduce.reduce(new Min(), summands), scalarType);
```

Na podoben način bomo v ukazno-pretokovnem delu namesto seštevanja delnih zmnožkov nad njimi izvajali minimum. To nam omogoča asociativnosti operacije minimum, ki je tudi eden izmed pogojev za polkolobar. Za poravnavo bomo 0 zamenjali za najvišjo možno vrednost (namesto $\infty$).

Ti dve prilagoditvi izvedeta eno množenje. Za končno rešitev je treba matriko spraviti na potenco najmanj $n-1$, kar naredimo z zaporednim kvadriranjem v ukazno-pretokovnem delu.

## 5.6 Eksperimentalno ovrednotenje

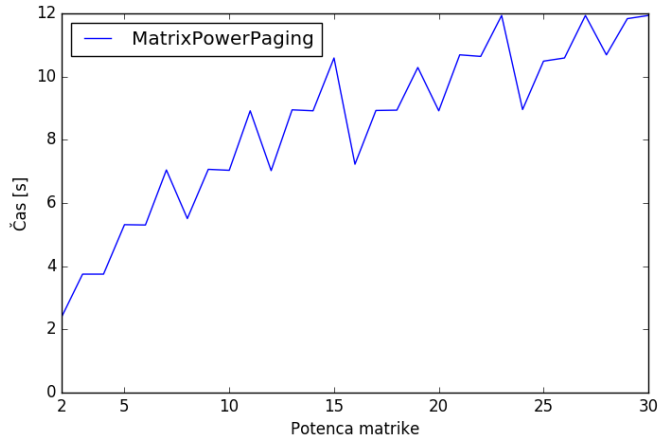
Teste smo izvajali na enaki programski in strojni opremi kot v razdelku 3.9. Testirali smo algoritem za potenciranje matrik in algoritem iskanja najkrajših poti med vsemi pari vozlišč.

### 5.6.1 Potenciranje matrik

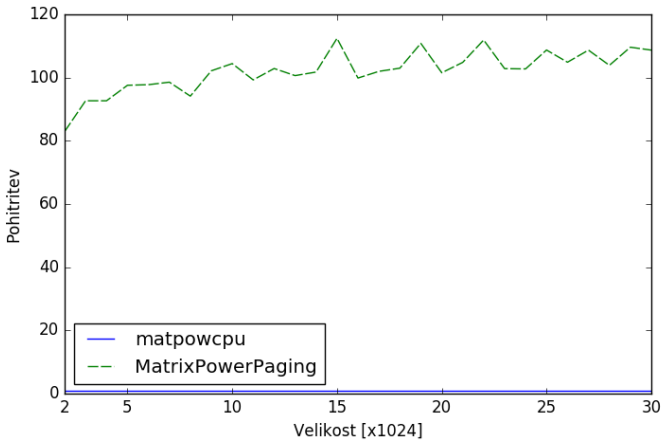
Slika 5.3 prikazuje graf časa izvajanja potenciranja za matriko velikosti 3072 x 3072 za potence od 2 do 30. Iz grafa je razvidno, da je čas odvisen od števila matričnih množenj, kar je tudi pričakovano. Slika 5.4 prikazuje faktor pohitritve podatkovno-pretokovnega algoritma v primerjavi z ukazno-pretokovnim. Podobno kot pri matričnem množenju z uporabo algoritma z razdelitvijo na ploščice v razdelku 4.8.2 so pohitritve okoli 100-kratne. To je bilo tudi pričakovano, saj je algoritem potenciranja pravzaprav zaporedje množenj, ki pa jih imata tako ukazno-pretokovna kot podatkovno-pretokovna implementacija enako.

### 5.6.2 Iskanje najkrajših poti med vsemi pari vozlišč

Čase izvajanja podatkovno-pretokovnega implementacije iskanja najkrajših poti med vsemi pari vozlišč smo primerjali z algoritmom Floyd-Warshall. Testna množica so bile kvadratne matrike velikosti od $512 \times 512$ do $6144 \times 6144$



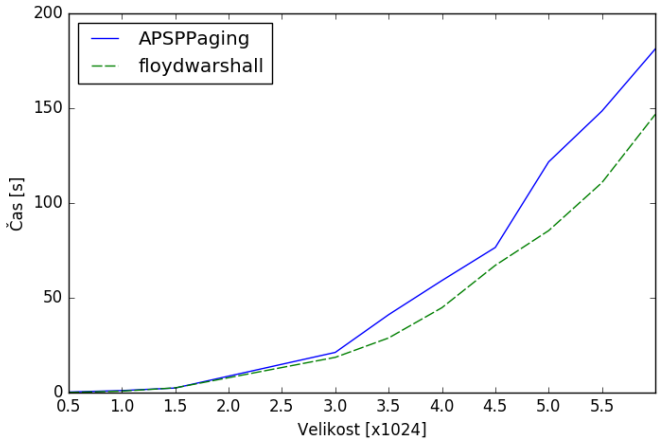

Slika 5.3: Čas izvajanja podatkovno-pretokovnega algoritma za potenciranje matrike velikosti 3072 × 3072




Slika 5.4: Pohitritev potenciranja za matriko velikosti 3072 × 3072

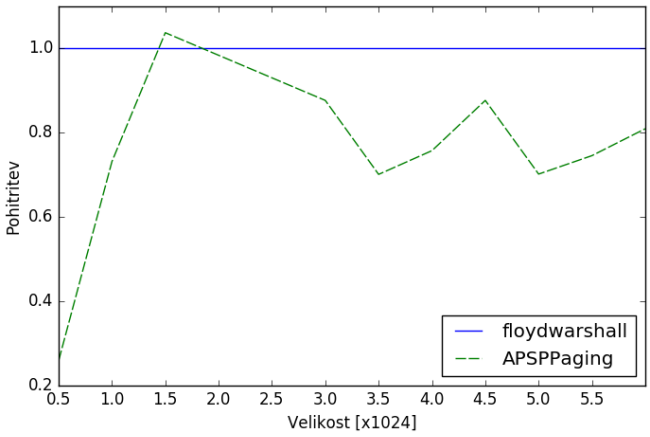
v korakih po 512. Slika 5.5 prikazuje primerjavo časa izvajanja. Na skoraj celotni množici se je algoritem Floyd-Warshall izkazal za nekoliko hitrejšega.

Bolj natančno primerjavo vidimo na sliki 5.6, ki prikazuje faktor pohitritve podatkovno-pretokovnega algoritma. Če zaradi majhnih časov zanemarimo prvi dve vrednosti (velikosti 512 in 1024), vidimo, da je na začetku podatkovno-pretokovni algoritem malenkost hitrejši. Za tem počasi izgublja




Slika 5.5: Prikaz časa izvajanja podatkovno-pretokovnega algoritma za iskanje vseh najkrajših poti skupaj z ukazno-pretokovno implementacijo algoritma Floyd-Warshall

prednost pred algoritmom Floyd-Warshall. Izkaže se, da je tokrat boljša asimptotična časovna zahtevnost prevladala pred prednostmi podatkovno-pretokovne implementacije. Kljub temu so rezultati zanimivi. Ker smo se osredotočali predvsem na hitrosti algoritmov podatkov, kot so poraba energije ter velikost opreme nimamo, jih tudi nismo upoštevali. Z uporabo metodologije, predlagane v [9], ki te parametre upošteva, bi bila lahko podatkovno-pretokovna rešitev konkurenčna algoritmu Floyd-Warshall. Poleg tega bi pri resničnem problemu program z lahkoto optimizirali na željeno natančnost, in s tem program pohitrili.




Slika 5.6: Faktor pohitritve podatkovno-pretokovnega algoritma za iskanje vseh najkrajših poti v primerjavi z ukazno-pretokovno implementacijo algoritma Floyd-Warshall

# Poglavje 6

# Sklepne ugotovitve

Podatkovno-pretokovna arhitektura kaže izjemen potencial v znanstvenem računanju. V prihodnosti se lahko njena uporaba zelo poveča, zato je vsak prispevek na tem področju še toliko bolj pomemben. V delu smo raziskali matrične algoritme na podatkovno-pretokovnem računalniku Maxeler. Uspešno smo implementirali več različic množenja matrike in vektorja ter matričnega množenja. Pri najboljšem algoritmih matričnega množenja smo doseglji več kot 100-kratne pohitritve v primerjavi z ukazno-pretokovno rešitvijo. Na podatkovno-pretokovno arhitekturo smo prenesli tudi algoritem iskanja najkrajših poti med vsemi pari vozlišč, ki pa je dosegal približno iste hitrosti kot ukazno-pretokovni algoritem Floyd-Warshall.

Če se najprej spomnimo, da je v računalništvu tudi 2-kratna pohitritev lahko velik dosežek, z lahkoto sklenemo, da so naše rešitve dobre. Med drugim naši rezultati pomenijo, da podatkovno-pretokovna arhitektura ni nekaj, kar bo uporabno šele v prihodnosti, ampak jo je za določene probleme smiselno uporabiti že danes. Pri iskanju najkrajših poti med vsemi pari vozlišč lahko sklenemo naslednje. Če nas zanimajo samo končni rezultati matrike cen, je morda bolje ostati pri ukazno-pretokovni implementaciji. Če nas zanimajo vmesni koraki (kar je pri tem in sorodnih problemih včasih potrebno), pa je prehod smiseln. Povemo lahko še, da je bila pri matričnem množenju operacija množenja kratka (samo eno množenje). Če bi bilo teh

operacij več, bi bile pohitritve morda še večje.

Izboljšave vidimo na naslednjih področjih. Algoritem z razdelitvijo na ploščice bi bilo mogoče nadgraditi z LMemom in akumulacijo celotne matrike prestaviti na čip. Kakšne pohitritve bi s tem dosegli, je v naprej težko napovedati, zato bi bilo treba algoritem nadgraditi in ga eksperimentalno ovrednotiti. Pri ostali algoritmih vidimo veliko potenciala predvsem v prilagoditvah za specifične probleme. Naši algoritmi se niso uporabljali v praksi, zaradi česar natančnosti predstavitve podatkov nismo spreminjali; bila je 32-bitna. Zelo zanimiv bi bil primer, kjer bi bil vhod matrika sosednosti in bi prilagodili natančnost števil na 1 bit. Algoritme bi bilo mogoče prevesti z veliko več cevmi oziroma z večjo velikostjo ploščice. Glede na to, da je bila pri algoritmu matričnega množenja z zmanjšanjem odmika glavna omejitev prepustnost, bi se morda v tem primeru odrezal najbolje in v primerjavi z ukazno-pretokovno različico dosegel velike pohitritve.

# Literatura


[1] D. Andrén, L. Hellström, K. Markström, Fast multiplication of matrices over a finitely generated semiring, Information Processing Letters 107 (6), 2008, str. 230–234.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Third Edition, The MIT Press, 2009, str. 594–602.

[3] U. Čibej, J. Mihelič, Adaptation and evaluation of the simplex algorithm for a data-flow architecture (osnutek), Advances in Computers, 2017.

[4] R. W. Floyd, Algorithm 97: Shortest path, Communications of the ACM 5 (6), 1962, str. 345.

[5] J. S. Golan, Semirings and their Applications, Springer Science & Business Media, 2013.

[6] J. Kepner, J. Gilbert, Graph Algorithms in the Language of Linear Algebra, Software, Environments, and Tools, Society for Industrial and Applied Mathematics, 2011.

[7] D. Kodek, Arhitektura in organizacija računalniških sistemov, Bi-tim, 2008, str. 51–54.

[8] D. König, Graphen und matrizen, Matematikai Lapok 38, 1931, str. 116–119.

[9] A. Kos, S. Tomažič, J. Salom, N. Trifunović, M. Valero, V. Milutinović, New benchmarking methodology and programming model for big data processing, International Journal of Distributed Sensor Networks 2015.

[10] F. Le Gall, Powers of tensors and fast matrix multiplication, v Proceedings of the 39th international symposium on symbolic and algebraic computation, ACM, 2014, str. 296–303.

[11] Maxeler Technologies, Acceleration tutorial loops and pipelining, 2015.

[12] Maxeler Technologies, Manager compiler tutorial, 2015.

[13] Maxeler Technologies, MaxCompiler, dostopno na: `https://www.maxeler.com/products/software/maxcompiler/` (pridobljeno: 1. 11. 2016).

[14] Maxeler Technologies, Maxcompiler white paper, dostopno na: `https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf` (pridobljeno: 1. 11. 2016).

[15] Maxeler Technologies, Maxpower, GitHub repository. Dostopno na: `https://github.com/maxeler/maxpower` (pridobljeno: 1. 11. 2016).

[16] Maxeler Technologies, Multiscale dataflow programming, 2015.

[17] Maxeler Technologies, Platforms, dostopno na: `https://www.maxeler.com/solutions/` (pridobljeno: 1. 11. 2016).

[18] C. C. McGeoch, A Guide to Experimental Algorithmics, 1st Edition, Cambridge University Press, New York, NY, USA, 2012.

[19] J. Mihelič, Trikotniki v grafu, dostopno na: `http://lalg.fri.uni-lj.si/jurij/blog/trikotniki-v-grafu/` (pridobljeno: 18. 12. 2016).

[20] J. Mihelič, U. Čibej, Experimental algorithmics for the Dataflow Architecutre: Guidelines and Issues, Transaction on Advanced Research 13 (1), 2017.

[21] A. Milinković, S. Milinković, L. Lazić, FPGA based dataflow accelerator for large matrix multiplication, 2013.

[22] V. Milutinović, J. Salom, N. Trifunović, R. Giorgi, Guide to DataFlow Supercomputing: Basic Concepts, Case Studies, and a Detailed Example, Springer, 2015.

[23] B. M. Moret, Towards a discipline of experimental algorithmics, Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges 59, 2002, str. 197–213.

[24] O. Pell, V. Averbukh, Maximum performance computing with dataflow engines, Computing in Science & Engineering 14 (4), 2012, str. 98–103.

[25] O. Pell, O. Mencer, Surviving the end of frequency scaling with reconfigurable dataflow computing, SIGARCH Computer Architecture News 39 (4), 2011, str. 60–65.

[26] B. Robič, T. Ungerer, Processor Architecture: From Dataflow to Superscalar and Beyond, 1st Edition, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999, predgovor in str. 55–57.

[27] J. Salom, H. A. Fujii, Selected HPC solutions Based on the Maxeler Data-Flow Approach, dostopno na: `http://home.etf.rs/~vm/os/vlsi/predavanja/Part%202%20Paper%201%20Jakob%20Salom%20Selected%20HPC%20Solutions%20Based%20on%20the%20Maxeler%20DataFlow%20Approach%20(2).pdf` (pridobljeno: 1. 11. 2016).

[28] S. G. Sedukhin, M. Paprzycki, Generalizing matrix multiplication for efficient computations on modern computers, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2011, str. 225–234.

[29] V. Simić, V. Cirić, N. Savić, I. Milentijević, Sparse Matrix Multiplication on Dataflow Engines, Springer International Publishing, Cham, 2016, str. 23–30.

[30] V. Strassen, Gaussian elimination is not optimal, Numerische Mathematik 13 (4), 1969, str. 354–156.

[31] B. Vilfan, Osnovni algoritmi, Fakulteta za računalništvo in informatiko, 1998, str. 147–150.

[32] N. Weaver, Y. Markovskiy, Y. Patel, J. Wawrzynek, Post-placement C-slow retiming for the Xilinx Virtex FPGA, in: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, ACM, 2002, str. 185–194.

[33] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, Y. Etsion, Hybrid dataflow/von-Neumann architectures, IEEE Transactions on Parallel and Distributed Systems 25 (6), 2014, str. 1489–1509.